11

Core String Edits, Alignments, and Dynamic Programming

11.1. Introduction

In this chapter we consider the inexact matching and alignment problems that form the core of the field of inexact matching and others that illustrate the most general techniques. Some of those problems and techniques will be further refined and extended in the next chapters. We start with a detailed examination of the most classic inexact matching problem solved by dynamic programming, the ***edit distance*** problem. The motivation for inexact matching (and, more generally, sequence comparison) in molecular biology will be a recurring theme explored throughout the rest of the book. We will discuss many specific examples of how string comparison and inexact matching are used in current molecular biology. However, to begin, we concentrate on the purely formal and technical aspects of defining and computing inexact matching.

11.2. The edit distance between two strings

Frequently, one wants a measure of the difference or ***distance*** between two strings (for example, in evolutionary, structural, or functional studies of biological strings; in textual database retrieval; or in spelling correction methods). There are several ways to formalize the notion of distance between strings. One common, and simple, formalization [389, 299], called ***edit distance***, focuses on *transforming* (or editing) one string into the other by a series of edit operations on individual characters. The permitted edit operations are ***insertion*** of a character into the first string, the ***deletion*** of a character from the first string, or the ***substitution*** (or ***replacement***) of a character in the first string with a character in the second string. For example, letting I denote the insert operation, D denote the delete operation, R the substitute (or replace) operation, and M the nonoperation of "match," then the string "***vintner***" can be edited to become "***writers***" as follows:

```
RIMDMDMMI
v intner
wri t ers
```

That is, v is replaced by w, r is inserted, i matches and is unchanged since it occurs in both strings, n is deleted, t is unchanged, n is deleted, er match and are unchanged, and finally s is inserted. We now more formally define edit transcripts and string transformations.

Definition A string over the alphabet I, D, R, M that describes a transformation of one string to another is called an ***edit transcript***, or transcript for short, of the two strings.

In general, given the two input strings S_1 and S_2, and given an edit transcript for S_1 and S_2, the transformation is accomplished by successively applying the specified operation in the transcript to the next character(s) in the appropriate string(s). In particular, let $next_1$ and

The final quote reflects the potential total impact on biology of the *first fact* and its exploitation in the form of sequence database searching. It is from an article [179] by Walter Gilbert, Nobel prize winner for the coinvention of a practical DNA sequencing method. Gilbert writes:

> **The new paradigm now emerging, is that all the 'genes' will be known (in the sense of being resident in databases available electronically), and that the starting point of biological investigation will be theoretical. An individual scientist will begin with a theoretical conjecture, only then turning to experiment to follow or test that hypothesis.**

Already, hundreds (if not thousands) of journal publications appear each year that report biological research where sequence comparison and/or database search is an integral part of the work. Many such examples that support and illustrate the *first fact* are distributed throughout the book. In particular, several in-depth examples are concentrated in Chapters 14 and 15 where multiple string comparison and database search are discussed. But before discussing those examples, we must first develop, in the next several chapters, the techniques used for approximate matching and (sub)sequence comparison.

Caveat

The *first fact of biological sequence analysis* is extremely powerful, and its importance will be further illustrated throughout the book. However, there is not a one-to-one correspondence between sequence and structure or sequence and function, because the converse of the *first fact* is not true. That is, high sequence similarity usually implies significant structural or functional similarity (the first fact), but structural or functional similarity does not necessarily imply sequence similarity, On the topic of protein structure, F. Cohen [106] writes "...similar sequences yield similar structures, but quite distinct sequences can produce remarkably similar structures". This *converse* issue is discussed in greater depth in Chapter 14, which focuses on multiple sequence comparison.

Another example of an alignment is shown on page 215 where ***vintner*** and ***writers*** are aligned with each other below their edit transcript. That example also suggests a duality between alignment and edit transcript that will be developed below.

Alignment versus edit transcript

From the mathematical standpoint, an alignment and an edit transcript are equivalent ways to describe a relationship between two strings. An alignment can be easily converted to the equivalent edit transcript and vice versa, as suggested by the ***vintner-writers*** example. Specifically, two opposing characters that mismatch in an alignment correspond to a substitution in the equivalent edit transcript; a space in an alignment contained in the first string corresponds in the transcript to an insertion of the opposing character into the first string; and a space in the second string corresponds to a deletion of the opposing character from the first string. Thus the edit distance of two strings is given by the alignment minimizing the number of opposing characters that mismatch plus the number of characters opposite spaces.

Although an alignment and an edit transcript are mathematically equivalent, from a modeling standpoint, an edit transcript is quite different from an alignment. An edit transcript emphasizes the putative ***mutational events*** (point mutations in the model so far) that transform one string to another, whereas an alignment only displays a relationship between the two strings. The distinction is one of ***process*** versus ***product.*** Different evolutionary models are formalized via different permitted string operations, and yet these can result in the same alignment. So an alignment alone blurs the mutational model. This is often a pedantic point but proves helpful in some discussions of evolutionary modeling.

We will switch between the language of edit transcript and alignment as is convenient. However, the language of alignment will often be preferred since it is more neutral, making no statement about process. And, the language of alignment will be more natural in the area of multiple sequence comparison.

11.3. Dynamic programming calculation of edit distance

We now turn to the algorithmic question of how to compute, via dynamic programming, the edit distance of two strings along with the accompanying edit transcript or alignment. The general paradigm of dynamic programming is probably well known to the readers of this book. However, because it is such a crucial tool and is used in so many string algorithms, it is worthwhile to explain in detail both the general dynamic programming approach and its specific application to the edit distance problem.

Definition For two strings S_1 and S_2, $D(i, j)$ is defined to be the edit distance of $S_1[1..i]$ and $S_2[1..j]$.

That is, $D(i, j)$ denotes the minimum number of edit operations needed to transform the first i characters of S_1 into the first j characters of S_2. Using this notation, if S_1 has n letters and S_2 has m letters, then the edit distance of S_1 and S_2 is ***precisely*** the value $D(n, m)$.

We will compute $D(n, m)$ by solving the more general problem of computing $D(i, j)$ for all combinations of i and j, where i ranges from zero to n and j ranges from zero to m. This is the standard ***dynamic programming*** approach used in a vast number of computational problems. The dynamic programming approach has three essential components – the ***recurrence relation,*** the ***tabular computation,*** and the ***traceback.*** We will explain each one in turn.

$next_2$ be pointers into S_1 and S_2. Both pointers begin with value one. The edit transcript is read and applied left to right. When symbol "I" is encountered, character $next_2$ is inserted before character $next_1$ in S_1, and pointer $next_2$ is incremented one character. When "D" is encountered, character $next_1$ is deleted from S_1 and $next_1$ is incremented by one character. When either symbol "R" or "M" is encountered, character $next_1$ in S_1 is replaced or matched by character *next2* from S_2, and then both pointers are incremented by one.

Definition The ***edit distance*** between two strings is defined as the minimum number of edit operations – insertions, deletions, and substitutions – needed to transform the first string into the second. For emphasis, note that matches are not counted.

Edit distance is sometimes referred to as ***Levenshtein distance*** in recognition of the paper [299] by V. Levenshtein where edit distance was probably first discussed.

We will sometimes refer to an edit transcript that uses the minimum number of edit operations as an ***optimal transcript.*** Note that there may be more than one optimal transcript. These will be called "cooptimal" transcripts when we want to emphasize the fact that there is more than one optimal.

The **edit distance problem** is to compute the edit distance between two given strings, along with an optimal edit transcript that describes the transformation.

The definition of edit distance implies that all operations are done to one string only. But edit distance is sometimes thought of as the minimum number of operations done on either of the two strings to transform both of them into a common third string. This view is equivalent to the above definition, since an insertion in one string can be viewed as a deletion in the other and vice versa.

11.2.1. String alignment

An edit transcript is a way to ***represent*** a particular transformation of one string to another. An alternate (and often preferred) way is by displaying an explicit ***alignment*** of the two strings.

Definition A (global) ***alignment*** of two strings S_1 and S_2 is obtained by first inserting chosen spaces (or dashes), either into or at the ends of S_1 and S_2, and then placing the two resulting strings one above the other so that every character or space in either string is opposite a unique character or a unique space in the other string.

The term "global" emphasizes the fact that for each string, the entire string is involved in the alignment. This will be contrasted with local alignment to be discussed later. Notice that our use of the word "alignment" is now much more precise than its use in Parts 1 and II. There, alignment was used in the colloquial sense to indicate how one string is placed relative to the other, and spaces were not then allowed in either string.

As an example of a global alignment, consider the alignment of the strings ***qacdbd*** and ***qawxb*** shown below:

$$\begin{array}{cccccc} q & a & c & d & b & d \\ q & a & w & x & \_ & b & \_ \end{array}$$

In this alignment, character c is mismatched with w, both the ds and the x are opposite spaces, and all other characters match their counterparts in the opposite string.

Since the last transcript symbol must either be I, D, *R*, or M, we have covered all cases and established the lemma. □

Now we look at the other side.

Lemma 11.3.2. $D(i, j) \leq \min[D(i-1, j)+1, D(i, j-1)+1, D(i-1, j-1)+t(i, j)]$.

PROOF The reasoning is very similar to that used in the previous lemma, but it achieves a somewhat different goal. The objective here is to demonstrate constructively the existence of transformations achieving each of the three values specified in the inequality. Then since all three values are feasible, their minimum is certainly feasible.

First, it is possible to transform $S_1[1..i]$ into $S_2[1..j]$ with exactly $D(i, j-1)+1$ edit operations. Simply transform $S_1[1..i]$ to $S_2[1..j-1]$ with the minimum number of edit operations, and then use one more to insert character $S_2(j)$ at the end. By definition, the number of edit operations in that particular way to transform S_1 to S_2 is exactly $D(i, j-1)+1$. Second, it is possible to transform $S_1[1..i]$ to $S_2[1..j]$ with exactly $D(i-1, j)+1$ edit operations. Transform $S_1[1..i-1]$ to $S_2[1..j]$ with the fewest operations, and then delete character $S_1(i)$. The number of edit operations in that particular transformation is exactly $D(i-1, j)+1$. Third, it is possible to do the transformation with exactly $D(i-1, j-1)+t(i, j)$ edit operations, using the same argument. □

Lemmas 11.3.1 and 11.3.2 immediately imply the correctness of the general recurrence relation for $D(i, j)$.

Theorem 11.3.1. When both i and j are strictly positive, $D(i, j) = \min[D(i-1, j)+1, D(i, j-1)+1, D(i-1, j-1)+t(i, j)]$.

PROOF Lemma 11.3.1 says that $D(i, j)$ must be equal to one of the three values $D(i-1, j)+1$, $D(i, j-1)+1$, or $D(i-1, j-1)+t(i, j)$. Lemma 11.3.2 says that $D(i, j)$ must be less than or equal to the smallest of those three values. It follows that $D(i, j)$ must therefore be equal to the smallest of those three values, and we have proven the theorem. □

This completes the first component of the dynamic programming method for edit distance, the recurrence relation.

11.3.2. Tabular computation of edit distance

The second essential component of any dynamic program is to use the recurrence relations to efficiently compute the value $D(n, m)$. We could easily code the recurrence relations and base conditions for $D(i, j)$ as a recursive computer procedure using any programming language that allows recursion. Then we could call that procedure with input m, n and sit back and wait for the answer.[1] This top-down recursive approach to evaluating $D(n, m)$ is simple to program but extremely inefficient for large values of n and m.

The problem is that the number of recursive calls grows exponentially with n and m (an easy exercise to establish). But there are only $(n+1) \times (m+1)$ combinations of i and j, so there are only $(n+1) \times (m+1)$ distinct recursive calls possible. Hence the inefficiency of the top-down approach is due to a massive number of redundant recursive calls to the procedure. A nice discussion of this phenomenon is contained in [112]. The key to a (vastly) more efficient computation of $D(n, m)$ is to abandon the simplicity of top-down computation and instead compute bottom-up.

[1] and wait, and wait, . . .

11.3.1. The recurrence relation

The recurrence relation establishes a recursive relationship between the value of $D(i, j)$, for i and j both positive, and values of D with index pairs smaller than i, j. When there are no smaller indices, the value of $D(i, j)$ must be stated explicitly in what are called the base conditions for $D(i, j)$.

For the edit distance problem, the base conditions are

$$D(i, 0) = i$$

and

$$D(0, j) = j.$$

The base condition $D(i, 0) = i$ is clearly correct (that is, it gives the number required by the definition of $D(i, 0)$) because the only way to transform the first i characters of S_1 to zero characters of S_2 is to delete all the i characters of S_1. Similarly, the condition $D(0, j) = j$ is correct because j characters must be inserted to convert zero characters of S_1 to j characters of S_2.

The recurrence relation for $D(i, j)$ when both i and j are strictly positive is

$$D(i, j) = \min[D(i-1, j) + 1, D(i, j-1) + 1, D(i-1, j-1) + t(i, j)],$$

where $t(i, j)$ is defined to have value 1 if $S_1(i) \neq S_2(j)$, and $t(i, j)$ has value 0 if $S_1(i) = S_2(j)$.

Correctness of the general recurrence

We establish correctness in the next two lemmas using the concept of an edit transcript.

Lemma 11.3.1. The value of $D(i, j)$ must be $D(i, j-1) + 1$, $D(i-1, j) + 1$, or $D(i-1, j-1) + t(i, j)$. There are no other possibilities.

PROOF Consider an edit transcript for the transformation of $S_1[1..i]$ to $S_2[1..j]$ using the minimum number of edit operations, and focus on the last symbol in that transcript. That last symbol must either be I, D, R, or M. If the last symbol is an I then the last edit operation is the insertion of character $S_2(j)$ onto the end of the (transformed) first string. It follows that the symbols in the transcript before that I must specify the minimum number of edit operations to transform $S_1[1..i]$ to $S_2[1..j-1]$ (if they didn't, then the specified transformation of $S_1[1..i]$ to $S_2[1..j]$ would use more than the minimum number of operations). By definition, that latter transformation takes $D(i, j-1)$ edit operations. Hence if the last symbol in the transcript is I, then $D(i, j) = D(i, j-1) + 1$.

Similarly, if the last symbol in the transcript is a D, then the last edit operation is the deletion of $S_1(i)$, and the symbols in the transcript to the left of that D must specify the minimum number of edit operations to transform $S_1[1..i-1]$ to $S_2[1..j]$. By definition, that latter transformation takes $D(i-1, j)$ edit operations. So if the last symbol in the transcript is D, then $D(i, j) = D(i-1, j) + 1$.

If the last symbol in the transcript is an R, then the last edit operation replaces $S_1(i)$ with $S_2(j)$, and the symbols to the left of R specify the minimum number of edit operations to transform $S_1[1..i-1]$ to $S_2[1..j-1]$. In that case $D(i, j) = D(i-1, j-1) + 1$. Finally, and by similar reasoning, if the last symbol in the transcript is an M, then $S_1(i) = S_2(j)$ and $D(i, j) = D(i-1, j-1)$. Using the variable $t(i, j)$ introduced earlier [i.e., that $t(i, j) = 0$ if $S_1(i) = S_2(j)$; otherwise $t(i, j) = 1$] we can combine these last two cases as one: If the last transcript symbol is R or M, then $D(i, j) = D(i-1, j-1) + t(i, j)$.

$D(i,j)$			*w*	*r*	*i*	*t*	*e*	*r*	*s*
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
v	1	1	1	2	3	4	5	6	7
i	2	2	2	2	2	3	4	5	6
n	3	3	3	3	3	3	4	5	6
t	4	4	4	4	4	*			
n	5	5							
e	6	6							
r	7	7							

Figure 11.2: Edit distances are filled in one row at a time, and in each row they are filled in from left to right. The example shows the edit distances $D(i, j)$ to column 3 of row 4. The next value to be computed is $D(4, 4)$, where an asterisk appears. The value for cell (4, 4) is 3, since $S_1(4) = S_2(4) = t$ and $D(3, 3) = 3$.

The reader should be able to establish that the table could also be filled in *columnwise* instead of rowwise, after row zero and column zero have been computed. That is, column one could be first filled in, followed by column two, etc. Similarly, it is possible to fill in the table by filling in successive anti-diagonals, We leave the details as an exercise.

11.3.3. The traceback

Once the value of the edit distance has been computed, how is the associated optimal edit transcript extracted? The easiest way (conceptually) is to establish pointers in the table as the table values are computed.

In particular, when the value of cell (i, j) is computed, set a pointer from cell (i, j) to cell $(i, j-1)$ if $D(i, j) = D(i, j-1) + 1$; set a pointer from (i,j) to $(i-1, j)$ if $D(i, j) = D(i-1, j) + 1$; and set a pointer from (i, j) to $(i-1, j-1)$ if $D(i, j) = D(i-1, j-1) + t(i, j)$. This rule applies to cells in row zero and column zero as well. Hence, for most objective functions, each cell in row zero points to the cell to its left, and each cell in column zero points to the cell just above it. For other cells, it is possible (and common) that more than one pointer is set from (i,j). Figure 11.3 shows an example.

The pointers allow easy recovery of an optimal edit transcript: Simply follow any path of pointers from cell (n, m) to cell $(0, 0)$. The edit transcript is recovered from the path by interpreting each horizontal edge in the path, from cell (i, j) to cell $(i, j-1)$, as an insertion (I) of character $S_2(j)$ into S_1; interpreting each vertical edge, from (i, j) to $(i-1, j)$, as a deletion (D) of $S_1(i)$ from S_1; and interpreting each diagonal edge, from (i, j) to $(i-1, j-1)$, as a match (M) if $S_1(i) = S_2(j)$ and as a substitution (R) if $S_1(i) \# S_2(j)$. That this traceback path specifies an optimal edit transcript can be proved in a manner similar to the way that the recurrences for edit distances were established. We leave this as an exercise.

Alternatively, in terms of aligning S_1 and S_2, each horizontal edge in the path specifies a space inserted into S_1, each vertical edge specifies a space inserted into S_2, and each diagonal edge specifies either a match or a mismatch, depending on the specific characters.

For example, there are three traceback paths from cell $(7, 7)$ to cell $(0, 0)$ in the example given in Figure 11.3. The paths are identical from cell $(7, 7)$ to cell $(3, 3)$, at which point

$D(i, j)$			*w*	*r*	*i*	*t*	*e*	*r*	*s*
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
v	1	1							
i	2	2							
n	3	3							
t	4	4							
n	5	5							
e	6	6							
r	7	7							

Figure 11.1: Table to be used to compute the edit distance between *vintner* and *writers*. The values in row zero and column zero are already included. They are given directly by the base conditions.

Bottom-up computation

In the bottom-up approach, we first compute $D(i, j)$ for the smallest possible values for i and j, and then compute values of $D(i, j)$ for increasing values of i and j. Typically, this bottom-up computation is organized with a dynamic programming table of size $(n + 1) \times (m + 1)$. The table holds the values of $D(i, j)$ for all the choices of i and j (see Figure 11.1). Note that string S_1 corresponds to the vertical axis of the table, while string S_2 corresponds to the horizontal axis. Because the ranges of i and j begin at zero, the table has a zero row and a zero column. The values in row zero and column zero are filled in directly from the base conditions for $D(i, j)$. After that, the remaining $n \times m$ subtable is filled in one row at time, in order of increasing i. Within each row, the cells are filled in order of increasing j.

To see how to fill in the subtable, note that by the general recurrence relation for $D(i, j)$, all the values needed for the computation of $D(1, 1)$ are known once $D(0, 0)$, $D(1, 0)$, and $D(0, 1)$ have been computed. Hence $D(1, I)$ can be computed after the zero row and zero column have been filled in. Then, again by the recurrence relations, after $D(1, 1)$ has been computed, all the values needed for the computation of $D(1, 2)$ are known. Following this idea, we see that the values for row one can be computed in order of increasing index j. After that, all the values needed to compute the values in row two are known, and that row can be filled in, in order of increasing j. By extension, the entire table can be filled in one row at a time, in order of increasing i, and in each row the values can be computed in order of increasing j (see Figure 11.2).

Time analysis

How much work is done by this approach? When computing the value for a specific cell (i, j), only cells $(i - 1, j - 1)$, $(i, j - 1)$, and $(i - 1, j)$ are examined, along with the two characters $S_1(i)$ and $S_2(j)$. Hence, to fill in one cell takes a constant number of cell examinations, arithmetic operations, and comparisons. There are $O(nm)$ cells in the table, so we obtain the following theorem.

Theorem 11.3.2. The dynamic programming table for computing the edit distance between *a* string *of length* n and *a* string of length m can *be filled* in with $O(nm)$ work. Hence, using dynamic programming, the edit distance $D(n, m)$ can be computed in $O(nm)$ time.

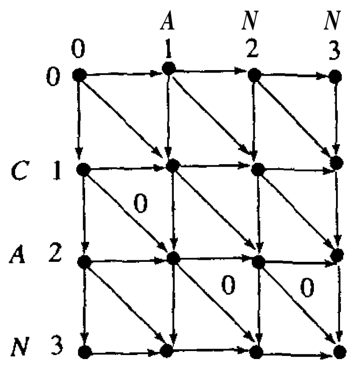

Figure 11.4: Edit graph for the strings CAN and *ANN*. The weight on each edge is one, except for the three zero-weight edges marked in the figure.

operations. Conversely, any optimal edit transcript *is* specified *by such* a path. Moreover; since a path describes only one transcript, the correspondence *between* paths and optimal transcripts is one-to-one.

The theorem can be proven by essentially the same reasoning that established the correctness of the recurrence relations for $D(i, j)$ and this is left to the reader. An alternative way to find the optimal edit transcript(s), without using pointers, is discussed in Exercise 9. Once the pointers have been established, all the cooptimal edit transcripts can be enumerated in $O(n + m)$ time per transcript. That is the focus of Exercise 12.

11.4. Edit graphs

It is often useful to represent dynamic programming solutions of string problems in terms of a weighted edir graph.

Definition Given two strings S_1 and S_2 of lengths n and m, respectively, a weighted edit graph has $(n + 1) \times (m + 1)$ nodes, each labeled with a distinct pair (i, j) $(0 \le i \le n, 0 \le j \le m)$. The specific edges and their edge weights depend on the specific string problem.

In the case of the edit distance problem, the edit graph contains a directed edge from each node (i, j) to each of the nodes $(i, j + 1)$, $(i + 1, j)$, and $(i + 1, j + 1)$, provided those nodes exist. The weight on the first two of these edges is one; the weight on the third (diagonal) edge is $t(i + 1, j + 1)$. Figure 11.4 shows the edit graph for strings CAN and ANN.

The central property of an edit graph is that any *shortest path* (one whose total weight is minimum) from start node $(0, 0)$ to destination node (n, m) specities an edit transcript with the minimum number of edit operations. Equivalently, any shortest path specifies a global alignment of minimum total weight. Moreover, the following theorem and corollary can be stated.

Theorem 11.4.1. An edit transcriptfor S_1, S_2 has the minimum number of edit operations *if and* only *if it* corresponds to a shortestpathfrom $(0, 0)$ to (n, m) in the edit graph.

Corollary 11.4.1. The set of all shortest paths from $(0, 0)$ to (n, m) in the edit graph exactly specifies the set of all optimal edit transcripts of S_1 to S_2. Equivalently, it specifies all the optimal (minimum weight) alignments of S_1 and S_2.

Viewing dynamic programming as a shortest path problem is often useful because there

$D(i, j)$			w	r	i	t	e	r	s
		0	1	2	3	4	5	6	7
	0	0	← 1	← 2	← 3	← 4	← 5	← 6	← 7
v	1	↑ 1	↖ 1	↖ ← 2	↖ ← 3	↖ ← 4	↖ ← 5	↖ ← 6	↖ ← 7
i	2	↑ 2	↖ ↑ 2	↖ 2	↖ 2	← 3	← 4	← 5	← 6
n	3	↑ 3	↖ ↑ 3	↖ ↑ 3	↖ ↑ 3	↖ 3	↖ ← 4	↖ ← 5	↖ ← 6
t	4	↑ 4	↖ ↑ 4	↖ ↑ 4	↖ ↑ 4	↖ 3	↖ ← 4	↖ ← 5	↖ ← 6
n	5	↑ 5	↖ ↑ 5	↖ ↑ 5	↖ ↑ 5	↑ 4	↖ 4	↖ ← 5	↖ ← 6
e	6	↑ 6	↖ ↑ 6	↖ ↑ 6	↖ ↑ 6	↑ 5	↖ 4	↖ ← 5	↖ ← 6
r	7	↑ 7	↖ ↑ 7	↖ 6	↖ ← ↑ 7	↑ 6	↑ 5	↖ 4	← 5

Figure 11.3: The complete dynamic programming table with pointers included. The arrow ← in cell (i, j) points to cell $(i, j-1)$, the arrow ↑ points to cell $(i-1, j)$, and the arrow ↖ points to cell $(i-1, j-1)$.

it is possible to either go up or to go diagonally. The three optimal alignments are:

```
w r i t _ e r s
v i n t n e r _
```

```
w   r i t e r s
v _ i n t n e r _
```

and

```
  w r i t e r s
_ v i n t n e r _
```

If there is more than one pointer from cell (n, m), then a path from (n, m) to (0, 0) can start with either of those pointers. Each of them is on a path from (n, m) to (0, 0). This property is repeated from any cell encountered. Hence a traceback path from (n, m) to (0, 0) can start simply by following any pointer out of (n, m); it can then be extended by following any pointer out of any cell encountered. Moreover, every cell except (0, 0) has a pointer out of it, so no path from (n, m) can get stuck. Since any path of pointers from (n, m) to (0, 0) specifies an optimal edit transcript or alignment, we have the following:

Theorem 11.3.3. *Once* the dynamic programming table with pointers has been computed, an optimal edit transcript can be found in $O(n + m)$ time.

We have now completely described the three crucial components of the general dynamic programming paradigm, as illustrated by the edit distance problem. We will later consider ways to increase the speed of the solution and decrease its needed space.

The pointers represent all optimal edit transcripts

The pointers that are built up while computing the values of the table do more than allow one optimal transcript (or optimal alignment) to be retrieved. They allow all optimal transcripts to be retrieved.

Theorem 11.3.4. Any path *from* (n, m) to (0, 0) following pointers established during the computation of $D(i, j)$ *specifies an edit transcript* with *the minimum number* of edit

The operation-weight edit distance problem can also be represented and solved as a shortest path problem on a weighted edit graph, where the edge weights correspond in the natural way to the weights of the edit operations. The details are straightforward and are thus left to the reader.

11.5.2. Alphabet-weight edit distance

Another critical, yet simple, generalization of edit distance is to allow the weight or score of a substitution to depend on exactly which character in the alphabet is being removed and which is being added. For example, it may be more costly to replace an *A* with a T than with a G. Similarly, we may want the weight of a deletion or insertion to depend on exactly which character in the alphabet is being deleted or inserted. We call this form of edit distance the ***alphabet-weight edit distance*** to distinguish it from the operation-weight edit distance problem.

The operation-weight edit distance problem is a special case of the alphabet-weight problem, and it is trivial to modify the previous recurrence relations (for operation-weight edit distance) to compute alphabet-weight edit distance. We leave that as an exercise. We will usually use the simple term ***weighted edit distance*** when we mean the alphabet-weight version. Notice that in weighted edit distance, the weight of an operation depends on what characters are involved in an operation but not on ***where*** those characters appear in the string.

When comparing proteins, "the edit distance" almost always means the alphabet-weight edit distance over the alphabet of amino acids. There is an extensive literature (and continuing research) on what scores should be used for operations on amino acid characters and how they should be determined. The dominant amino acid scoring schemes are now the PAM matrices of Dayhoff [122] and the newer BLOSUM scoring matrices of the Henikoffs [222], although these matrices are actually defined in terms of a maximization problem (similarity) rather than edit distance.[3] Recently, a mathematical theory has been developed [16, 262] concerning the way scores should be interpreted and how a scoring scheme should relate both to the data it is obtained from and to the types of searches it is designed for. We will briefly discuss this issue again in Section 15.11.2.

When comparing DNA strings, unweighted or operation-weight edit distance is more often computed. For example, the popular database searching program, BLAST, scores identities as $+5$ and mismatches as -4. However, alphabet-weighted edit distance is also of interest and alphabet-based scoring schemes for DNA have been suggested (for example see [252]).

11.6. String similarity

Edit distance is one of the ways that the relatedness of two strings has been formalized. An alternate, and often preferred, way of formalizing the relatedness of two strings is to measure their *similarity* rather than their distance. This approach is chosen in most biological applications for technical reasons that should be clear later. When focusing on

[3] **In a pure computer science or mathematical discussion of alphabet-weight edit distance, we would prefer to use the general term "weight matrix" for the matrix holding the alphabet-dependent substitution scores. However, molecular biologists use the terms "amino acid substitution matrix" or "nucleotide substitution matrix" for those matrices, and they use the term "weight matrix" for a very different object (See Section 14.3.1). Therefore, to maintain generality. and yet to keep in some harmony with the molecular biology literature, we will use the general term "scoring matrix".**

are many tools for investigating and compactly representing shortest paths in graphs. This view will be exploited in Section 13.2 when suboptimal solutions are discussed.

11.5. Weighted edit distance

11.5.1. Operation weights

An easy, yet crucial, generalization of edit distance is to allow an arbitrary ***weight*** or ***cost*** or ***score***[2] to be associated with every edit operation, as well as with a match. Thus, any insertion or deletion has a weight denoted d, a substitution has a weight r, and a match has a weight e (which is usually small compared to the other weights and is often zero). Equivalently, an ***operation-weight alignment*** is one where each mismatch costs r, each match costs e, and each space costs d.

Definition With arbitrary operation weights, the ***operation-weight edit distance problem*** is to find an edit transcript that transforms string S_1 into S_2 with the minimum total operation weight.

In these terms, the edit distance problem we have considered so far is just the problem of finding the minimum operation-weight edit transcript when $d = 1$, $r = 1$, and $e = 0$. But, for example, if each mismatch has a weight of 2, each space has a weight of 4, and each match a weight of 1, then the alignment

$$\begin{array}{cccccccc} w & r & i & t & e & r & s \\ v & i & n & t & n & e & r & - \end{array}$$

has a total weight of 17 and is an optimal alignment.

Because the objective function is to minimize total weight and because a substitution can be achieved by a deletion followed by an insertion, if substitutions are to be allowed then a substitution weight should be less than the sum of the weights for a deletion plus an insertion.

Computing operation-weight edit distance

The operation-weight edit distance problem for two strings of length n and m can be solved in $O(nm)$ time by a minor extension of the recurrences for edit distance. $D(i, j)$ now denotes the minimum total weight for edit operations transforming $S_1[1..i]$ to $S_2[1..j]$. We again use $t(i, j)$ to handle both substitution and equality, where now $t(i, j) = e$ if $S_1(i) = S_2(j)$; otherwise $t(i, j) = r$. Then the base conditions are

$$D(i, 0) = i \times d$$

and

$$D(0, j) = j \times d.$$

The general recurrence is

$$D(i, j) = \min[D(i, j-1) + d, D(i-1, j) + d, D(i-1, j-1) + t(i, j)].$$

[2] The terms "weight" or "cost" are heavily used in the computer science literature, while the term "score" is used in the biological literature. We will use these terms more or less interchangeably in discussing algorithms, but the term "score" will be used when talking about specific biological applications,

Definition $V(i, j)$ is defined as the *value* of the optimal alignment of prefixes $S_1[1..i]$ and $S_2[1..j]$.

Recall that a dash ("–") is used to represent a space inserted into a string. The base conditions are

$$V(0, j) = \sum_{1 \le k \le j} s(\_, S_2(k))$$

and

$$V(i, 0) = \sum_{1 \le k \le i} s(S_1(k), \_).$$

For i and j both strictly positive, the general recurrence is

$$V(i, j) = \max[V(i-1, j-1) + s(S_1(i), S_2(j)), V(i-1, j) + s(S_1(i), \_),$$
$$V(i, j-1) + s(\_, S_2(j))].$$

The correctness of this recurrence is established by arguments similar to those used for edit distance. In particular, in any alignment $\mathcal{A}$, there are three possibilities: characters $S_1(i)$ and $S_2(j)$ are in the same position (opposite each other), $S_1(i)$ is in a position after $S_2(j)$, or $S_1(i)$ is in a position before $S_2(j)$. The correctness of the recurrence is based on that case analysis. Details are left to the reader.

If S_1 and S_2 are of length n and m, respectively, then the value of their optimal alignment is given by $V(n,m)$. That value, and the entire dynamic programming table, can be obtained in $O(nm)$ time, since only three comparisons and arithmetic operations are needed per cell. By leaving pointers while filling in the table, as was done with edit distance, an optimal alignment can be constructed by following any path of pointers from cell (n,m) to cell $(0, 0)$. So the optimal (global) alignment problem can be solved in $O(nm)$ time, the same time as for edit distance.

11.6.2. Special cases of similarity

By choosing an appropriate scoring scheme, many problems can be modeled as special cases of optimal alignment or similarity. One important example is the ***longest common subsequence problem.***

Definition In a string S, a ***subsequence*** is defined as a subset of the characters of S arranged in their original "relative" order. More formally, a subsequence of a string S of length n is specified by a list of indices $i_1 < i_2 < i_3 < \ldots < i_k$, for some $k \le n$. The subsequence specified by this list of indices is the string $S(i_1)S(i_2)S(i_3)\ldots S(i_k)$.

To emphasize again, a ***subsequence*** need not consist of contiguous characters in S, whereas the characters of a ***substring*** must be contiguous.[4] Of course, a substring satisfies the definition for a subsequence. For example, "its" is a subsequence of "winters" but not a substring, whereas "inter" is both a substring and a subsequence.

Definition Given two strings S_1 and S_2, a ***common subsequence*** is a subsequence that appears both in S_1 and S_2. The ***longest common subsequence problem*** is to find a longest common subsequence (*lcs*) of S_1 and S_2.

[4] The distinction between subsequence and substring is often lost in the biological literature. But algorithms for substrings are usually quite different in spirit and efficiency than algorithms for subsequences, so the distinction is an important one.

similarity, the language of alignment is usually more convenient than the language of edit transcript. We now begin to develop a precise definition of similarity.

Definition Let Σ be the alphabet used for strings S_1 and S_2, and let C' be C with the added character "_" denoting a space. Then, for any two characters x, y in Σ', $s(x, y)$ denotes the value (or score) obtained by aligning character x against character y.

Definition For a given alignment A of S_1 and S_2, let S_1' and S_2' denote the strings after the chosen insertion of spaces, and let 1 denote the (equal) length of the two strings S_1' and S_2' in A. The value of alignment A is defined as $\sum_{i=1}^{l} s(S_1'(i), S_2'(i))$.

That is, every position i in $\mathcal{A}$ specifies a pair of opposing characters in the alphabet Σ', and the value of $\mathcal{A}$ is obtained by summing the value contributed by each pair.

For example, let $\Sigma = (a, b, c, d\}$ and let the pairwise scores be defined in the following matrix:

s	*a*	*b*	*c*	*d*	_
a	1	−1	−2	0	−1
b		3	−2	−1	0
c			0	−4	−2
d				3	−1
_					0

Then the alignment

$$\begin{array}{ccccccc} c & a & c & \_ & d & b & d \\ c & a & b & b & d & b & \_ \end{array}$$

has a total value of $0 + 1 - 2 + 0 + 3 + 3 - 1 = 4$.

In string similarity problems, scoring matrices usually set $s(x, y)$ to be greater than or equal to zero if characters **x**, y of Σ' match and less than zero if they mismatch. With such a scoring scheme, one seeks an alignment with as large a value as possible. That alignment will emphasize matches (or similarities) between the two strings while penalizing mismatches or inserted spaces. Of course, the meaningfulness of the resulting alignment may depend heavily on the scoring scheme used and how match scores compare to mismatch and space scores. Numerous character-pair scoring matrices have been suggested for proteins and for DNA [81, 122, 127,222,252,4001, and no single scheme is right for all applications. We will return to this issue in Sections 13.1, 15.7, and 15.10.

Definition Given a pairwise scoring matrix over the alphabet C', the *similarity* of two strings S_1 and S_2 is defined as the value of the alignment A of S_1 and S_2 that maximizes total alignment value. This is also called the optimal alignment value of S_1 and S_2.

String similarity is clearly related to alphabet-weight edit distance, and depending on the specific scoring matrix involved, one can often transform one problem into the other. An important difference between similarity and weighted edit distance will become clear in Section 11.7, after we discuss local alignment.

11.6.1. Computing similarity

The similarity of two strings S_1 and S_2, and the associated optimal alignment, can be computed by dynamic programming with recurrences that should by now be very intuitive.

One example where end-spaces should be free is in the shotgun sequence assembly (see Sections 16.14 and 16.15). In this problem, one has a large set of partially overlapping substrings that come from many copies of one original but unknown string; the problem is to use comparisons of pairs of substrings to infer the correct original string. Two random substrings from the set are unlikely to be neighbors in the original string, and this is reflected by a low end-space free alignment score for those two substrings. But if two substrings do overlap in the original string, then a "good-sized" suffix of one should align to a "good-sized" prefix of the other with only a small number of spaces and mismatches (reflecting a small percentage of sequencing errors). This overlap is detected by an end-space free weighted alignment with high score. Similarly the case when one substring contains another can be detected in this way. The procedure for deducing candidate neighbor pairs is thus to compute the end-space free alignment between every pair of substrings; those pairs with high scores are then the best candidates. We will return to shotgun sequencing and extend this discussion in Part IV, Section 16.14.

To implement free end spaces in computing similarity, use the recurrences for global alignment (where all spaces count) detailed on page 227, but change the base conditions to $V(i,0) = V(0,j) = 0$, for every i and j. That takes care of any spaces on the left end of the alignment. Then fill in the table as in the case of global alignment. However, unlike global alignment, the value of the optimal alignment is not necessarily found in cell (n, m). Rather, the value of the optimal alignment with free ends is the maximum value over all cells in row n or column m. Cells in row n correspond to alignments where the last character of string S_1 contributes to the value of the alignment, but characters of S_2 to its right do not. Those characters are opposite end spaces, which are free. Cells in column m have a similar characterization. Clearly, optimal alignment with free end spaces is solved in $O(nm)$ time, the same time as for global alignment.

11.6.5. Approximate occurrences of P in T

We now examine another important variant of global alignment.

Definition Given a parameter 6, a substring T' of T is said to be an approximate occurrence of P if and only if the optimal alignment of P to T' has value at least δ.

The problem of determining if there is an approximate occurrence of P in T is an important and natural generalization of the exact matching problem. It can be solved as follows: Use the same recurrences (given on page 227) as for global alignment between P and T and change only the base condition for $V(0, j)$ to $V(0, j) = 0$ for all j. Then fill in the table (leaving the standard backpointers). Using this variant of global alignment, the following theorem can be proved.

Theorem 11.6.2. There is an approximate occurrence of P in T ending at *position* j of T *if and* only *if* $V(n, j) \geq 6$. Moreover; T[k..j] is an approximate occurrence of P *in* T *if* and only if $V(n, j) \geq 6$ and there is a path of backpointers from cell (n, j) to cell $(0, k)$.

Clearly, the table can be filled in using $O(nm)$ time, but if all approximate occurrence of P in T are to be explicitly output, then $\Theta(nm)$ time may not be sufficient. A sensible compromise is to identify every position j in T such that $V(n, j) \geq 6$, and then for each such j, explicitly output only the shortest approximate occurrence of P that ends at position j. That substring T' is found by traversing the backpointers from (n, j) until a

The lcs problem is important in its own right, and we will discuss some of its uses and some ideas for improving its computation in Section 12.5. For now we show that it can be modeled and solved as an optimal alignment problem.

Theorem 11.6.1. With a scoring scheme that scores *a* onefor *each* match and a zero *for* each mismatch o rspace, the matched characters in an alignment of maximum valueform a longest common subsequence.

The proof is immediate and is left to the reader. It follows that the longest common subsequence of strings of lengths n and m, respectively, can be computed in $O(nm)$ time.

At this point we see the first of many differences between substring and subsequence problems and why it is important to clearly distinguish between them. In Section 7.4 we established that the longest common substring could be found in O(n + m) time, whereas here the bound established for finding longest common subsequence is O(n x m) (although this bound can be reduced somewhat). This is typical – substring and subsequence problems are generally solved by different methods and have different time and space complexities.

11.6.3. Alignment graphs for similarity

As was the case for edit distance, the computation of similarity can be viewed as a path problem on a directed acyclic graph called an alignment graph. The graph is the same as the edit graph considered earlier, but the weights on the edges are the specific values for aligning a specific pair of characters or a character against a space. The start node of the alignment graph is again the node associated with cell (0, 0), and the destination node is associated with cell (n, m) of the dynamic programming table, but the optimal alignment comes from the longest start to destination path rather than from the shortest path. It is again true that the longest paths in the alignment graph are in one-to-one correspondence with the optimal (maximum value) alignments. In general, computing longest paths in graphs is difficult, but for directed acyclic graphs the longest path is found in time proportional to the number of edges in the graph, using a variant of dynamic programming (which should come as no surprise). Hence for alignment graphs, the longest path can be found in $O(nm)$ time.

11.6.4. End-space free variant

There is a commonly used variant of string alignment called end-spacefree alignment. In this variant, any spaces at the end or the beginning of the alignment contribute a weight of zero, no matter what weight other spaces contribute. For example, in the alignment

$$\begin{array}{ccccccccc} \_ & \_ & c & a & c & \_ & d & b & d \\ l & t & c & a & b & b & d & b & \_ \end{array}$$

the two spaces at the left end of the alignment are free, as is the single space at the right end.

Making end spaces free in the objective function encourages one string to align in the interior of the other, or the suffix of one string to align with a prefix of the other. This is desirable when one believes that those kinds of alignments reflect the "true" relationship of the two strings. Without a mechanism to encourage such alignments, the optimal alignment might have quite a different shape and not capture the desired relationship.

strings may be related. When comparing protein sequences, local alignment is also critical because proteins from very different families are often made up of the same structural or functional subunits (motifs or domains), and local alignment is appropriate in searching for these (unknown) subunits. Similarly, different proteins are often made from related motifs that form the inner core of the protein, but the motifs are separated by outside surface looping regions that can be quite different in different proteins.

A very interesting example of conserved domains comes from the proteins encoded by ***homeobox*** genes. Homeobox genes [319, 381] show up in a wide variety of species, from fruit flies to frogs to humans. These genes regulate high-level embryonic development, and a single mutation in these genes can transform one body part into another (one of the original mutation experiments causes fruit fly antenna to develop as legs, but it doesn't seem to bother the fly very much). The protein sequences that these genes encode are very different in each species, except in one region called the ***homeodomain.*** The homeodomain consists of about sixty amino acids that form the part of the regulatory protein that binds to DNA. Oddly, homeodomains made by certain insect and mammalian genes are particularly similar, showing about 50 to 95% identity in alignments without spaces. Protein-to-DNA binding is central in how those proteins regulate embryo development and cell differentiation. So the amino acid sequence in the most biologically critical part of those proteins is highly conserved, whereas the other parts of the protein sequences show very little similarity. In cases such as these, local alignment is certainly a more appropriate way to compare protein sequences than is global alignment.

Local alignment in protein is additionally important because particular isolated characters of related proteins may be more highly conserved than the rest of the protein (for example, the amino acids at the ***active*** site of an enzyme or the amino acids in the ***hydrophobic core*** of a globular protein are the most highly conserved). Local alignment will more likely detect these conserved characters than will global alignment. A good example is the family of ***serine proteases*** where a few isolated, conserved amino acids characterize the family. Another example comes from the Helix-Turn-Helix motif, which occurs frequently in proteins that regulate DNA transcription by binding to DNA. The tenth position of the Helix-Turn-Helix motif is very frequently occupied by the amino acid glycine, but the rest of the motif is more variable.

The following quote from C. Chothia [101] further emphasizes the biological importance of protein domains and hence of local string comparison.

> Extant proteins have been produced from the original set *not just* by point mutations, insertions and deletions but also by combinations of genes to give chimeric proteins. This is particularly true of the very large proteins produced in the recent stages of evolution. Many of these are built of different combinations of protein domains that have been selected from a relatively small repertoire.

Doolittle [129] summarizes the point: "The underlying message is that one must be alert to regions of similarity even when they occur embedded in an overall background of dissimilarity."

Thus, the dominant viewpoint today is that local alignment is the most appropriate type of alignment for comparing proteins from different protein families. However, it has also been pointed out [359, 360] that one often sees extensive global similarity in pairs of protein strings that are first recognized as being related by strong local similarity. There are also suggestions [316] that in some situations global alignment is more effective than local alignment in exposing important biological commonalities.

cell in row zero is reached, breaking ties by choosing a vertical pointer over a diagonal one and a diagonal one over a horizontal one.

11.7. Local alignment: finding substrings of high similarity

In many applications, two strings may not be highly similar in their entirety but may contain regions that are highly similar. The task is to find and extract a pair of regions, one from each of the two given strings, that exhibit high similarity. This is called the ***local alignment*** or ***local similarity problem*** and is defined formally below.

Local alignment problem Given two strings S_1 and S_2, find substrings α and β of S_1 and S_2, respectively, whose similarity (optimal global alignment value) is maximum over all pairs of substrings from S_1 and S_2. We use v^* to denote the value of an optimal solution to the local alignment problem.

For example, consider the strings $S_1 = pqraxabcsrvq$ and $S_2 = xyaxbacsll$. If we give each match a value of 2, each mismatch a value of -2, and each space a value of -1, then the two substrings $\alpha = axabcs$ and $\beta = axbacs$ of S_1 and S_2, respectively, have the following optimal (global) alignment

$$\begin{array}{ccccccc} a & x & a & b & \_ & c & s \\ a & x & \_ & b & a & c & s \end{array}$$

which has a value of 8. Furthermore, over all choices of pairs of substrings, one from each of the two strings, those two substrings have maximum similarity (for the chosen scoring scheme). Hence, for that scoring scheme, the optimal local alignment of S_1 and S_2 has value 8 and is defined by substrings $axabcs$ and $axbacs$.

It should be clear why local alignment is defined in terms of similarity, which maximizes an objective function, rather than in terms of edit distance, which minimizes an objective. When one seeks a pair of substrings to minimize distance, the optimal pairs would be exactly matching substrings under most natural scoring schemes. But the matching substrings might be just a single character long and would not identify a region of high similarity. A formulation such as local alignment, where matches contribute positively and mismatches and spaces contribute negatively, is more likely to find more meaningful regions of high similarity.

Why local alignment?

Global alignment of protein sequences is often meaningful when the two strings are members of the same protein family. For example, the protein *cytochrome c* has almost the same length in most organisms that produce it, and one expects to see a relationship between two cytochromes from any two different species over the entire length of the two strings. The same is true of proteins in the *globin* family, such as *myoglobin* and ***hemoglobin***. In these cases, global alignment is meaningful. When trying to deduce evolutionary history by examining protein sequence similarities and differences, one usually compares proteins in the same sequence family, and so global alignment is typically meaningful and effective in those applications.

However, in many biological applications, local similarity (local alignment) is far more meaningful than global similarity (global alignment). This is particularly true when long stretches of anonymous DNA are compared, since only ***some*** internal sections of those

Theorem 11.7.2. *If i', j' is an index pair maximizing $v(i, j)$ over all i, j pairs, then a pair of substrings solving the local suffix alignment problem for i', j' also solves the local alignment problem.*

Thus a solution to the local suffix alignment problem solves the local alignment problem. We now turn our attention to the problem of finding $\max[v(i, j) : i \leq n, j \leq m]$ and a pair of strings whose alignment has maximum value.

11.7.2. How to solve the local suffix alignment problem

First, $v(i, 0) = 0$ and $v(0, j) = 0$ for all i, j, since we can always choose an empty suffix.

Theorem 11.7.3. *For $i > 0$ and $j > 0$, the proper recurrence for $v(i, j)$ is*

$$v(i, j) = \max[0, v(i-1, j-1) + s(S_1(i), S_2(j)),$$
$$v(i-1, j) + s(S_1(i), \_), v(i, j-1) + s(\_, S_2(j))].$$

PROOF The argument is similar to the justifications of previous recurrence relations. Let α and β be the substrings of S_1 and S_2 whose global alignment establishes the optimal local alignment. Since α and β are permitted to be empty suffixes of $S_1[1..i]$ and $S_2[1..j]$, it is correct to include 0 as a *candidate* value for $v(i, j)$. However, if the optimal α is not empty, then character $S_1(i)$ must either be aligned with a space or with character $S_2(j)$. Similarly, if the optimal β is not empty, then $S_2(j)$ is aligned with a space or with $S_1(i)$. So we justify the recurrence based on the way characters $S_1(i)$ and $S_2(j)$ may be aligned in the optimal local suffix alignment for i, j.

If $S_1(i)$ is aligned with $S_2(j)$ in the optimal local i, j suffix alignment, then those two characters contribute $s(S_1(i), S_2(j))$ to $v(i, j)$, and the remainder of $v(i, j)$ is determined by the local suffix alignment for indices $i-1, j-1$. That local suffix alignment must be optimal and so has value $v(i-1, j-1)$. Therefore, if $S_1(i)$ and $S_2(j)$ are aligned with each other, $v(i, j) = v(i-1, j-1) + s(S_1(i), S_2(j))$.

If $S_1(i)$ is aligned with a space, then by similar reasoning $v(i, j) = v(i-1, j) + s(S_1(i), -)$, and if $S_2(j)$ is aligned with a space then $v(i, j) = v(i, j-1) + s(\_, S_2(j))$. Since all cases are exhausted, we have proven that $v(i, j)$ must either be zero or be equal to one of the three other terms in the recurrence.

On the other hand, for each of the four terms in the recurrence, there *is* a way to choose suffixes of $S_1[1..i]$ and $S_2[1..j]$ so that an alignment of those two suffixes has the value given by the associated term. Hence the optimal suffix alignment value is at least the maximum of the four terms in the recurrence. Having proved that $v(i, j)$ must be one of the four terms, and that it must be greater than or equal to the maximum of the four terms, it follows that $v(i, j)$ must be equal to the maximum which proves the theorem. □

The recurrences for local suffix alignment are almost identical to those for global alignment. The only difference is the inclusion of zero in the case of local suffix alignment. This makes intuitive sense. In both global alignment and local suffix alignment of prefixes $S_1[1..i]$ and $S_2[1..j]$ the end characters of any alignment are specified, but in the case of local suffix alignment, any number of initial characters can be ignored. The zero in the recurrence implements this, acting to "restart" the recurrence.

Given Theorem 11.7.2, the method to compute v^* is to compute the dynamic programming table for $v(i, j)$ and then find the largest value in *any* cell in the table, say in cell (i^*, j^*). As usual, pointers are created while filling in the values of the table. After cell

11.7.1. Computing local alignment

Why not look for regions of high similarity in two strings by first globaly aligning those strings? A global alignment between two long strings will certainly be influenced by regions of high similarity, and an optimal global alignment might well align those corresponding regions with each other. But more often, local regions of high local similarity would get lost in the overall optimal global alignment. Therefore, to identify high local similarity it is more effective to search explicitly for local similarity.

We will show that if the lengths of strings S_1 and S_2 are n and m, respectively, then the local alignment problem can be solved in $O(nm)$ time, the same time as for global alignment. This efficiency is surprising because there are $\Theta(n^2m^2)$ pairs of substrings, so even if a global alignment could be computed in constant time for each chosen pair, the time bound would be $\Theta(n^2m^2)$. In fact, if we naively use $O(kl)$ for the bound on the time to align strings of lengths k and I, then the resulting time bound for the local alignment problem would be $O(n^3m^3)$, instead of the $O(nm)$ bound that we will establish. The $O(nm)$ time bound was obtained by Temple Smith and Michael Waterman [411] using the algorithm we will describe below.

In the definition of local alignment given earlier, any scoring scheme was permitted for the global alignment of two chosen substrings. One slight restriction will help in computing local alignment. We assume that the global alignment of two empty strings has value zero. That assumption is used to allow the local alignment algorithm to choose two empty substrings for a and β. Before describing the solution to the local alignment problem. it will be helpful to consider first a more restricted version of the problem.

Definition Given a pair of indices i $\leq$ n and j $\leq$ m, the local *suffix* alignment problem is to find a (possibly empty) suffix α of $S_1[1..i]$ and a (possibly empty) suffix β of $S_2[1..j]$ such that $V(\alpha, \beta)$ is the maximum over all pairs of suffixes of $S_1[1..i]$ and $S_2[1..j]$. We use $v(i, j)$ to denote the value of the optimal local suffix alignment for the given index pair i. j.

For example, suppose the objective function counts 2 for each match and -1 for each mismatch or space. If S_1 = abcxdex and S_2 = xxxcde, then $v(3, 4) = 2$ (the two cs match), $v(4, 5) = 1$ (cx aligns with *cd*), $v(5, 5) = 3$ (x–d aligns with *xcd*), and $v(6, 6) = 5$ (*x_de* aligns with xcde).

Since the definition allows either or both of the suffixes to be empty, $v(i, j)$ is always greater than or equal to zero.

The following theorem shows the relationship between the local alignment problem and the local suffix alignment problem. Recall that v^* is the value of the optimal local alignment for two strings of length n and m.

Theorem 11.7.1. $v^* = \max[v(i, j) : i \leq n, j \leq m]$.

PROOF Certainly $v^* \geq \max[v(i, j) : i \leq n, j \leq m]$, because the optimal solution to the local suffix alignment problem for any i, j is a feasible solution to the local alignment problem. Conversely, let a, β be the substrings in an optimal solution to the local alignment problem and suppose a ends at position i* and β ends at j* Then α, β also defines a local suffix alignment for index pair i^*, j* and so $v^* \leq v(i^*, j^*) \leq \max[v(i, j) : i \leq n, j \leq m]$, and both directions of the lemma are established. □

Theorem 11.7.1 only specifies the value v^*, but its proof makes clear how to find substrings whose alignment have that value. In particular,

```
c t t t a a c - - a - a c
c - - - c a c c c a t - c
```

Figure 11.5: An alignment with seven spaces distributed into four gaps.

with similarity (global alignment value) of $v(i, j)$. Thus, an easy way to look for a set of highly similar substrings is to find a set of cells in the table with a value above some set threshold. Not all similar substrings will be identified in this way, but this approach is common in practice.

The need for good scoring schemes

The utility of optimal local alignment is affected by the scoring scheme used. For example, if matches are scored as one, and mismatches and spaces as zero, then the optimal local alignment will be determined by the longest common ***subsequence.*** Conversely, if mismatches and spaces are given large negative scores, and each match is given a score of one, then the optimal local alignment will be the longest common ***substring.*** In most cases, neither of these is the local alignment of interest and some care is required to find an application-dependent scoring scheme that yields meaningful local alignments. For local alignment, the entries in the scoring matrix must have an average score that is negative. Otherwise the resulting "local" optimal alignment tends to be a global alignment. Recently, several authors have developed a rather elegant theory of what scoring schemes for local alignment mean in the context of database search and how they should be derived. We will briefly discuss this theory in Section 15.11.2.

11.8. Gaps

11.8.1. Introduction to Gaps

Until now the central constructs used to measure the value of an alignment (and to define similarity) have been ***matches, mismatches,*** and ***spaces.*** Now we introduce another important construct, ***gaps.*** Gaps help create alignments that better conform to underlying biological models and more closely fit patterns that one expects to find in meaningful alignments.

Definition A gap is any *maximal,* ***consecutive run*** of spaces in a ***single*** string of a given alignment.[5]

A gap may begin before the start of S, in which case it is bordered on the right by the first character of S, or it may begin after the end of S, in which case it is bordered on the left by the last character of S. Otherwise, a gap must be bordered on both sides by characters of S. A gap may be as small as a single space. As an example of gaps, consider the alignment in Figure 11.5, which has four gaps containing a total of seven spaces. That alignment would be described as having five matches, one mismatch, four gaps, and seven spaces. Notice that the last space in the first string is followed by a space in the second string, but those two spaces are in two gaps and do not form a single gap.

By including a term in the objective function that reflects the gaps in the alignment one has some influence on the ***distribution*** of spaces in an alignment and hence on the overall shape of the alignment. In the simplest objective function that includes gaps,

[5] **Sometimes in the biology literature the term "space" (as we use it) is not used. Rather, the term "gap" is used both for "space" and for "gap" (as we have defined it here). This can cause much confusion, and in this book the terms "gap" and "space" have distinct meanings.**

(i^*, j^*) is found, the substrings α and β giving the optimal local alignment of S_1 and S_2 are found by tracing back the pointers from cell (i^*, j^*) until an entry (i', j') is reached that has value zero. Then the optimal local alignment substrings are $\alpha = S_1[i'..i^*]$ and $\beta = S_2[j'..j^*]$.

Time analysis

Since it takes only four comparisons and three arithmetic operations per cell to compute $v(i, j)$, it takes only $O(nm)$ time to fill in the entire table. The search for v^* and the traceback clearly require only $O(nm)$ time as well, so we have established the following desired theorem:

Theorem 11.7.4. ***For two strings S_1 and S_2 of lengths n and m, the local alignment problem can be solved in $O(nm)$ time, the same time as for global alignment.***

Recall that the pointers in the dynamic programming table for edit distance, global alignment, and similarity encode all the optimal alignments. Similarly, the pointers in the dynamic programming table for local alignment encode the optimal local alignments as follows.

Theorem 11.7.5. ***All optimal local alignments of two strings are represented in the dynamic programming table for $v(i, j)$ and can be found by tracing any pointers back from any cell with value v^*.***

We leave the proof as an exercise.

11.7.3. Three final comments on local alignment

Terminology for local and global alignment

In the biological literature, global alignment (similarity) is often referred to as a Needleman–Wunsch [347] alignment after the authors who first discussed global similarity. Local alignment is often referred to as a Smith–Waterman [411] alignment after the authors who introduced local alignment. There is, however, some confusion in the literature between Needleman–Wunsch and Smith–Waterman as ***problem statements*** and as ***solution methods.*** The original solution given by Needleman–Wunsch runs in cubic time and is rarely used. Hence "Needleman–Wunsch" usually refers to the global alignment ***problem.*** The Smith–Waterman method runs in quadratic time and is commonly used, so "Smith–Waterman" often refers to their specific solution as well as to the problem statement. But there are solution methods to the (Smith–Waterman) local alignment problem that differ from the Smith–Waterman solution and yet are sometimes also referred to as "Smith–Waterman".

Using Smith–Waterman to find several regions of high similarity

Very often in biological applications it is not sufficient to find just a single pair of substrings of input strings of S_1 and S_2 with the optimal local alignment. Rather, what is required is to find all or "many" pairs of substrings that have similarity above some threshold. A specific application of this kind will be discussed in Section 18.2, and the general problem will be studied much more deeply in Section 13.2. Here we simply point out that, in practice, the dynamic programming table used to solve the local suffix alignment problem is often used to find additional pairs of substrings with "high" similarity. The key observation is that for any cell (i, j) in the table, one can find a pair of substrings of S_1 and S_2 (by traceback)

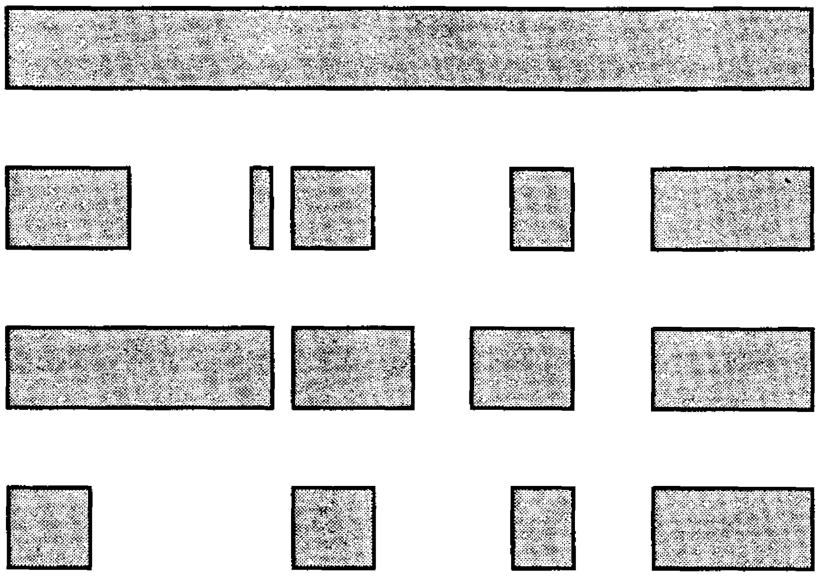

Figure 11.6: Each of the four rows represents part of the RNA sequence of one strain of the **HIV-1** virus. The HIV virus mutates rapidly, so that mutations can be observed and traced. The bottom three rows are from virus strains that have each mutated from an ancestral strain represented in the top row. Each of the bottom sequences is shown aligned to the top sequence. A dark box represents a substring that matches the corresponding substring in the top sequence, while each white space represents a gap resulting from a known sequence deletion. This figure is adapted from one in [123].

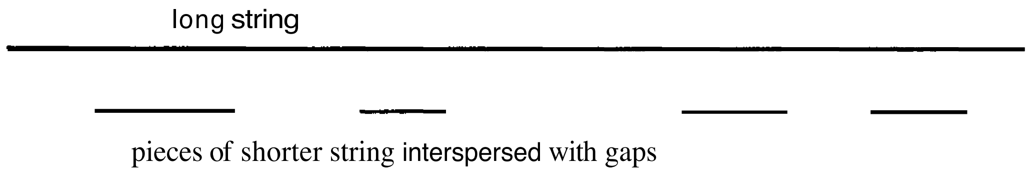

Figure 11.7: In cDNA matching, one expects the alignment of the smaller string with the longer string to consist of a few regions of very high similarity, interspersed with relatively long gaps.

shows up as a gap when two proteins are aligned. In some contexts, many biologists consider the proper identification of the major (long) gaps as *the* essential problem of protein alignment. If the long (major) gaps have been selected correctly, the rest of the alignment – reflecting point mutations – is then relatively easy to obtain.

An alignment of two strings is intended to reflect the cost (or likelihood) of mutational events needed to transform one string to another. Since a gap of more than one space can be created by a single mutational event, the alignment model should reflect the true distribution of spaces into gaps, not merely the number of spaces in the alignment. It follows that the model must specify how to weight gaps so as to reflect their biological meaning. In this chapter we will discuss different proposed schemes for weighting gaps, and in later chapters we will discuss additional issues in scoring gaps. First we consider a concrete example illustrating the utility of the gap concept.

11.8.3. cDNA matching: a concrete illustration

One concrete illustration of the use of gaps in the alignment model comes from the problem of *cDNA matching*. In this problem, one string is much longer than the other, and the alignment best reflecting their relationship should consist of a few regions of very high similarity interspersed with "long" gaps in the shorter string (see Figure 11.7). Note that the matching regions can have mismatches and spaces, but these should be a small percentage of the region.

each gap contributes a constant weight W_g, independent of how long the gap is. That is, each individual space is free, so that $s(x, \_) = s(\_, x) = 0$ for every character x. Using the notation established in Section 11.6, (page 226), we write the value of an alignment containing k gaps as

$$\sum_{i=1}^{l} s(S_1'(i), S_2'(i)) - kW_g.$$

Changing the value of W_g relative to the other weights in the objective function can change how spaces are distributed in the optimal alignment. A large W_g encourages the alignment to have few gaps, and the aligned portions of the two strings will fall into a few substrings. A smaller W_g allows more fragmented alignments. The influence of W_g on the alignment will be discussed more deeply in Section 13.1.

11.8.2. Why gaps?

Most of the biological justifications given for the importance of local alignment (see Section 11.7) apply as well to justify the gap as an explicit concept in string alignment.

Just as a space in an alignment corresponds to an insertion or deletion of a single character in the edit transcript, a gap in string S_1 opposite substring a in string S_2 corresponds to either a deletion of cr from S_1 or to an insertion of α into S_2. The concept of a gap in an alignment is therefore important in many biological applications because the insertion or deletion of an entire substring (particularly in DNA) often occurs as single mutational event. Moreover, many of these single mutational events can create gaps of quite varying sizes with almost equal likelihood (within a wide, but bounded, range of sizes). Much of the repetitive DNA discussed in Section 7.11.1 is caused by single mutational events that copy and insert long pieces of DNA. Other mutational mechanisms that make long insertions or deletions in DNA include: unequal crossing-over in meiosis (causing an insertion in one string and a reciprocal deletion in the other); DNA slippage during replication (where a portion of the DNA is repeated on the replicated copy because the replication machinery loses its place on the template, slipping backwards and repeating a section); insertion of transposable elements (jumping genes) into a DNA string; insertions of DNA by retroviruses; and *translocations* of DNA between chromosomes [301, 317]. See Figure 11.6 for an example of gaps in genomic sequence data.

When computing alignments for the purpose of deducing evolutionary history over a long period of time, it is often the gaps that are the most informative part of the alignments. In DNA strings, single character substitutions due to point mutations occur continuously and usually at a much faster rate than (nonfatal) mutational events causing gaps. The analogous gene (specifying the "same" protein) in two species can thus be very different at the DNA sequence level, making it difficult to sort out evolutionary relationships on the basis of string similarity (without gaps). But large insertions and deletions in molecules that show up as gaps in alignments occur less frequently than substitutions. Therefore, common gaps in pairs of aligned strings can sometimes be the key features used to deduce the overall evolutionary history of a set of strings [45, 405]. Later, in Section 17.3.2, we will see that such gaps can be considered as evolutionary characters in certain approaches to building evolutionary trees.

At the protein level, recall that many proteins are "built of different combinations of protein domains that have been selected from a relatively small repertoire"[101]. Hence two protein strings might be relatively similar over several intervals but differ in intervals where one contains aprotein domain that the other does not. Such an interval most naturally

Certainly, you don't want to set a large penalty for spaces, since that would align all the cDNA string close together, rather than allowing gaps in the alignment corresponding to the long introns. You would also want a rather high penalty for mismatches. Although there may be a few sequencing errors in the data, so that some mismatches will occur even when the cDNA is properly cut up to match the exons, there should not be a large percentage of mismatches. In summary, you want small penalties for spaces, relatively large penalties for mismatches, and positive values for matches.

What kind of alignment would likely result using an objective function that has low space penalty, high mismatch penalty, positive match value of course, and no term for gaps? Remember that the long string contains more than one gene, that the exons are separated by long introns, and that DNA has an alphabet of only four letters present in roughly equal amounts. Under these conditions, the optimal alignment would probably be the ***longest common subsequence*** between the short cDNA string and the long anonymous DNA string. And because the introns are long and DNA has only four characters, that common subsequence would likely match *all* of the characters in the cDNA. Moreover, because of small but real sequencing errors, the true alignment of the cDNA to its exons would not match all the characters. Hence the longest common subsequence would likely have a higher score than the correct alignment of the cDNA to exons. But the longest common subsequence would fragment the cDNA string over the longer DNA and not give an alignment of the desired form – it would not pick out its exons.

Putting a term for gaps in the objective function rectifies the problem. By adding a constant gap weight W_g for each gap in the alignment, and setting W_g appropriately (by experimenting with different values of W_g), the optimal alignment can be induced to cut up the cDNA to match its exons in the longer string.[6] As before, the space penalty is set to zero, the match value is positive, and the mismatch penalty is set high.

Processed pseudogenes

A more difficult version of cDNA matching arises in searching anonymous DNA for ***processed pseudogenes.*** A pseudogene is a near copy of a working gene that has mutated sufficiently from the original copy so that it can no longer function. Pseudogenes are very common in eukaryotic organisms and may play an important evolutionary role, providing a ready pool of diverse "near genes". Following the view that new genes are created by the process of ***duplication with modification*** of existing genes [127, 128, 130], pseudogenes either represent trial genes that failed or future genes that will function after additional mutations.

A pseudogene may be located very far from the gene it corresponds to, even on a different chromosome entirely, but it will usually contain both the introns and the exons derived from its working relative. The problem of finding pseudogenes in anonymous sequenced DNA is therefore related to that of finding repeated substrings in a very long string.

A more interesting type of pseudogene, the ***processed pseudogene,*** contains only the exon substrings from its originating gene. Like cDNA, the introns have been removed and the exons concatenated. It is thought that a processed pseudogene originates as an mRNA that is retranscribed back into DNA (by the enzyme Reverse Transcriptase) and inserted into the genome at a random location.

Now, given a long string of anonymous DNA that might contain both a processed pseudogene and its working ancestor, how could the processed pseudogenes be located?

[6] This really works, and it is a very instructive exercise to try it out empirically.

Biological setting of the problem

In eukaryotes, a gene that codes for a protein is typically made up of alternating *exons* (expressed sequences), which contribute to the code for the protein, and introns (intervening sequences), which do not. The number of exons (and hence also introns) is generally modest (four to twenty say), but the lengths of the introns can be huge compared to the lengths of the exons.

At a very coarse level, the protein specified by a eukaryotic gene is made in the following steps. First, an RNA molecule is transcribedfrom the DNA of the gene. That RNA transcript is a complement of the DNA in the gene in that each A in the gene is replaced by *U* (uracil) in the RNA, each T is replaced by A, each *C* by G, and each G by C. Moreover, the RNA transcript covers the entire gene, introns as well as exons. Then, in a process that is not completely understood, each intron-exon boundary in the transcript is located, the RNA corresponding to the introns is spliced out (or *snurped* out by a molecular complex called a *snrp* [420]), and the RNA regions corresponding to exons are concatenated. Additional processing occurs that we will not describe. The resulting RNA molecule is called the messenger RNA (mRNA); it leaves the cell nucleus and is used to create the protein it encodes.

Each cell (usually) contains a copy of all the chromosomes and hence of all the genes of the entire individual, yet in each specialized cell (a liver cell for example) only a small fraction of the genes are expressed. That is, only a small fraction of the proteins encoded in the genome are actually produced in that specialized cell. A standard method to determine which proteins are expressed in the specialized cell line, and to hunt for the location of the encoding genes, involves capturing the mRNA in that cell after it leaves the cell nucleus. That mRNA is then used to create a DNA string complementary to it. This string is called cDNA (complementary DNA). Compared to the original gene, the cDNA string consists only of the concatenation of exons in the gene.

It is routine to capture mRNA and make cDNA libraries (complete collections of a cell's mRNA) for specific cell lines of interest. As more libraries are built up, one collects a reflection of all the genes in the genome and a taxonomy of the cells that the genes are expressed in. In fact, a major component of the Human Genome Project [111], [399] is to obtain cDNAs reflecting most of the genes in the human genome. This effort is also being conducted by several private companies and has led to some interesting disputes over patenting cDNA sequences.

After cDNA is obtained, the problem is to determine where the gene associated with that cDNA resides. Presently, this problem is most often addressed with laboratory methods. However, if the cDNA is sequenced or partially sequenced (and in the Human Genome Project, for example, the intent is to sequence parts of each of the obtained cDNAs), and if one has sequenced the part of the genome containing the gene associated with that cDNA (as, for example, one would have after sequencing the entire genome), then the problem of finding the gene site given a cDNA sequence becomes a string problem. It becomes one of aligning the cDNA string against the longer string of sequenced DNA in a way that reveals the exons. It becomes the cDNA matching problem discussed above.

Why gaps are needed in the objective function

If the objective function includes terms only for matches, mismatches, and spaces, there seems no way to encourage the optimal alignment to be of the desired form. It's worth a moment's effort to explain why.

The alphabet-weight version of the affine gap weight model again sets $s(x, \_) = s(\_, x) = 0$ and has the objective of finding an alignment to

$$\text{maximize}\left(\sum_{i=1}^{l}[s(S_1'(i), S_2'(i))] - W_g(\#gaps) - W_s(\#spaces)\right).$$

The affine gap weight model is probably the most commonly used gap model in the molecular biology literature, although there is considerable disagreement about what W_g and W_s should be [161] (in addition to questions about W_m and W_{ms}). For aligning amino acid strings, the widely used search program FASTA [359] has chosen the default settings of $W_g = 10$ and $W_s = 2$. We will return to the question of the choice of these settings in Section 13.1.

It has been suggested [57, 183, 466] that some biological phenomena are better modeled by a gap weight function where each additional space in a gap contributes less to the gap weight than the preceding space (a function with negative second derivative). In other words, a gap weight that is a *convex*,[8] but not affine, function of its length. An example is the function $W_g + \log_e q$, where q is the length of the gap. Some biologists have suggested that a gap function that initially increases to a maximum value and then decreases to near zero would reflect a combination of different biological phenomena that insert or delete DNA.

Finally, the most general gap weight we will consider is the ***arbitrary gap weight***, where the weight of a gap is an arbitrary function $w(q)$ of its length q. The constant, affine, and convex weight models are of course subcases of the arbitrary weight model.

Time bounds for gap choices

As might be expected, the time needed to optimally solve the alignment problem with arbitrary gap weights is greater than for the other models. In the case that $w(q)$ is a totally arbitrary function of gap length, the optimal alignment can be found in $O(nm^2 + n^2m)$ time, where n and $m \geq n$ are the lengths of the two strings. In the case that $w(q)$ is convex, we will show that the time can be reduced to $O(nm \log m)$ (a further reduction is possible, but the algorithm is much too complex for our interests). In the affine (and hence constant) case the time bound is $O(nm)$, which is the same time bound established for the alignment model without the concept of gaps. In the next sections we will first discuss alignment for arbitrary gap weights and then show how to reduce the running time for the case of affine weight functions. The $O(nm \log m)$-time algorithm for convex weights is more complex than the others and is deferred until Chapter 13.

11.8.5. Arbitrary gap weights

This case was first introduced and solved in the classic paper of Needleman and Wunsch [347], although with somewhat different detail and terminology than used here.

For arbitrary gap weights, we will develop recurrences that are similar to (but more detailed than) the ones used in Section 11.6.1 for optimal alignment without gaps. There is, however, a subtle question about whether these recurrences correctly model the biologist's view of gaps. We will examine that issue in Exercise 45.

To align strings S_1 and S_2, consider, as usual, the prefixes $S_1[1..i]$ of S_1 and $S_2[1..j]$ of S_2. Any alignment of those two prefixes is one of the following three types (see Figure 11.8):

[8] Some call this *concave*.

The problem is similar to cDNA matching but more difficult because one does not have the cDNA in hand. We leave it to the reader to explore the use of repeat finding methods, local alignment, and gap weight selection in tackling this problem.

Caveat

The problems of cDNA and pseudogene matching illustrate the utility of including gaps in the alignment objective function and the importance of weighting the gaps appropriately. It should be noted, however, that in practice one can approach these matching problems by a judicious use of local alignment without gaps. The idea is that in computing local alignment, one can find not only the most similar pair of substrings but many other highly similar pairs of substrings (see Sections 13.2.4, and 11.7.3). In the context of cDNA or pseudogene matching, these pairs will likely be the exons, and so the needed match of cDNA to exons can be pieced together from a number of nonoverlapping local alignments. This is the more typical approach in practice.

11.8.4. Choices for gap weights

As illustrated by the example of cDNA matching, the appropriate use of gaps in the objective function aids in the discovery of alignments that satisfy an expected shape. But clearly, the way gaps are weighted critically influences the effectiveness of the gap concept. We will examine in detail four general types of gap weights: constant, *affine*, convex, and arbitrary.

The simplest choice is the constant gap weight introduced earlier, where each individual space is free, and each gap is given a weight of W_g independent of the number of spaces in the gap. Letting W_m and W_{ms} denote weights for matches and mismatches, respectively, the operator-weight version of the problem is:

Find an alignment $\mathcal{A}$ to maximize $[W_m(\#\text{ matches}) - W_{ms}(\#\text{ mismatches}) - W_g(\#\ gaps)]$.

More generally, if we adopt the alphabet-dependent weights for matches and mismatches, the objective in the constant gap weight model is:

Find an alignment A to maximize $\left(\sum_{i=1}^{l}[s(S_1'(i), S_2'(i))] - W_g(\#\, gaps)\right)$,

where $s(x, \_) = s(\_, x) = 0$ for every character x, and S_1' and S_2' represent the strings S_1 and S_2 after insertion of spaces.

A generalization of the constant gap weight model is to add a weight W_s for each space in the gap. In this case, W_g is called the gap initiation weight because it can represent the cost of starting a gap, and W_s is called the gap extension weight because it can represent the cost of extending the gap by one space. Then the operator-weight version of the problem is:

Find an alignment to maximize $[W_m(\#\text{ matches}) - W_{ms}(\#\ mismatches) - W_g(\#\ gaps) - W_s(\#\text{ spaces})]$.

This is called the *affine* gap weight model[7] because the weight contributed by a single gap of length q is given by the affine function $W_g + q$ W,. The constant pap weight model is simply the affine model with $W_s = 0$.

[7] The affine gap model is sometimes called the *linear* weight model, and I prefer that term. However, "affine" has become the dominant term in the biological literature, and "linear" there usually refers to an affine function with $W_g = 0$.

where $G(0,0) = 0$, but $G(i, j)$ is undefined when exactly one of i or j is zero. Note that $V(0,0) = w(0)$, which will most naturally be assigned to be zero.

When end spaces, and hence end gaps, are free, then the optimal alignment value is the maximum value over any cell in row n or column m, and the base cases are

$$V(i, 0) = 0,$$

$$V(0, j) = 0.$$

Time analysis

Theorem 11.8.1. Assuming that $|S_1| = n$ and $|S_2| = m$, the recurrences can be evaluated in $O(nm^2 + n^2m)$ time.

PROOF We evaluate the recurrences by the usual approach of filling in an $(n+1) \times (m+1)$ size table one row at time, where each row is filled from left to right. For any cell (i, j), the algorithm examines one other cell to evaluate $G(i, j)$, j cells of row i to evaluate $E(i, j)$, and i cells of column j to evaluate $F(i, j)$. Therefore, for any fixed row, $m(m+1)/2 = \Theta(m^2)$ cells are examined to evaluate all the E values in that row, and for any fixed column, $\Theta(n^2)$ cells are examined to evaluate all the F values of that column. The theorem then follows since there are n rows and m columns. □

The increase in running time over the previous case ($O(nm)$ time when gaps are not in the model) is caused by the need to look j cells to the left and i cells above to determine $V(i, j)$. Before gaps were included in the model, $V(i, j)$ depended only on the three cells adjacent to (i, j), and so each $V(i, j)$ value was computed in constant time. We will show next how to reduce the number of cell examinations for the case of affine gap weights; later we will show a more complex reduction for the case of convex gap weights.

11.8.6. Affine (and constant) gap weights

Here we examine in detail the simplest affine gap weight model and show that optimal alignments in that model can be computed in $O(nm)$ time. That bound is the same as for the alignment model without a gap term in the objective function. So although an explicit gap term in the objective function makes the alignment model much richer, it does not increase the running time used (in an asymptotic, worst-case sense) to find an optimal alignment. This important result was derived by several different authors (e.g., [18], [166], [186]). The same result then holds immediately for constant gap weights.

Recall that the objective is to find an alignment to

$$\text{maximize}[W_m(\#\,matches) - W_{ms}(\#\,mismatches) - W_g(\#\,gaps) - W_s(\#\,\text{spaces})].$$

We will use the same variables $V(i, j)$, $E(i, j)$, $F(i, j)$, and $G(i, j)$ used in the recurrences for arbitrary gap weights. The definition and meanings of these variables remain unchanged, but the recurrence relations will be modified for the case of affine gap weights.

The key insight leading to greater efficiency in the affine gap case is that the increase in the total weight of a gap contributed by each additional space is a constant W_s independent of the size of the gap to that point. In other words, in the affine gap weight model $w(q+1) - w(q) = W_s$ for any gap length q greater than zero. This is in contrast to the arbitrary weight case where there is no predictable relationship between $w(q)$ and $w(q+1)$. Because the gap weight increases by the same W_s for each space after the first one, when evaluating $E(i, j)$ or $F(i, j)$ we need not be concerned with exactly where a gap begins, but only whether it

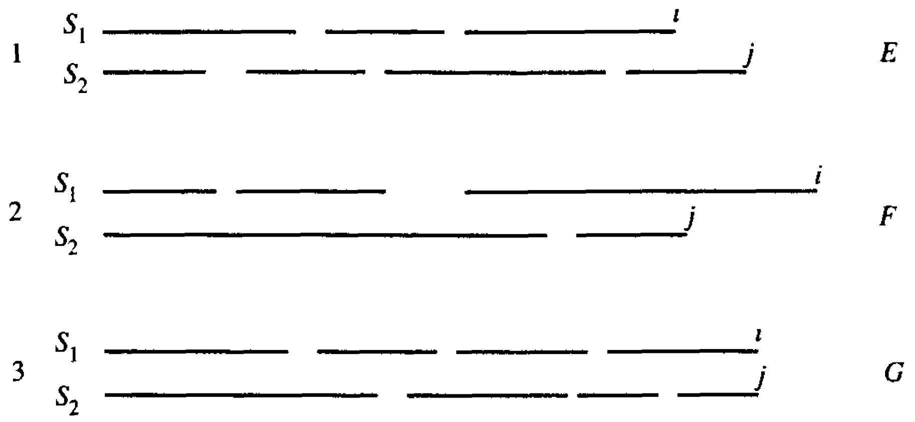

Figure 11.8: The recurrences for alignment with gaps are divided into three types of alignments: 1. those that align $S_1(i)$ to the left of $S_2(j)$, 2. those that align $S_1(i)$ to the right of $S_2(j)$, and 3. those that align them opposite each other.

1. Alignments of $S_1[1..i]$ and $S_2[1..j]$ where character $S_1(i)$ is aligned to a character strictly to the left of character $S_2(j)$. Therefore, the alignment ends with a gap in S_1.
2. Alignments of the two prefixes where $S_1(i)$ is aligned strictly to the right of $S_2(j)$. Therefore, the alignment ends with a gap in S_2.
3. Alignments of the two prefixes where characters $S_1(i)$ and $S_2(j)$ are aligned opposite each other. This includes both the case that $S_1(i) = S_2(j)$ and that $S_1(i) \# S_2(j)$.

Clearly, these three types of alignments cover all the possibilities.

Definition Define $E(i, j)$ as the maximum value of any alignment of type 1; define $F(i, j)$ as the maximum value of any alignment of type 2; define $G(i, j)$ as the maximum value of any alignment of type 3; and finally define $V(i, j)$ as the maximum value of the three terms $E(i, j), F(i, j), G(i, j)$.

Recurrences for the case of arbitrary gap weights

By dividing the types of alignments into three cases, as above, we can write the following recurrences that establish $V(i, j)$:

$$V(i,j) = \max[E(i,j), F(i,j), G(i,j)],$$

$$G(i,j) = V(i-1, j-1) + s(S_1(i), S_2(j)),$$

$$E(i,j) = \max_{0 \le k \le j-1} [V(i,k) - w(j-k)],$$

$$F(i,j) = \max_{0 \le l \le i-1} [V(l,j) - w(i-l)].$$

To complete the recurrences, we need to specify the base cases and where the optimal alignment value is found. If all spaces are included in the objective function, even spaces that begin or end an alignment, then the optimal value for the alignment is found in cell (n, m), and the base case is

$$V(i,0) = -w(i),$$

$$V(0,j) = -w(j),$$

$$E(i,0) = -w(i),$$

$$F(0,j) = -w(j),$$

11.9, Exercises

1. Write down the edit transcript for the alignment example on page 226.
2. The definition given in this book for string transformation and edit distance allows at most one operation per position in each string. But part of the motivation for string transformation and edit distance comes from an attempt to model evolution, where there is no restriction on the number of mutations that could occur at the same position. A deletion followed by an insertion and then a replacement could all happen at the same position. However, even though multiple operations at the same position are allowed, they will not occur in the transformation that uses the fewest number of operations. Prove this.
3. In the discussion of edit distance, all transforming operations were assumed to be done to one string only, and a "hand-waiving" argument was given to show that no greater generality is gained by allowing operations on both strings. Explain in detail why there is no loss in generality in restricting operations to one string only.
4. Give the details for how the dynamic programming table for edit distance or alignment can be filled in columnwise or by successive antidiagonals. The antidiagonal case is useful in the context of practical parallel computation. Explain this.
5. In Section 11.3.3, we described how to create an edit transcript from the traceback path through the dynamic programming table for edit distance. Prove that the edit transcript created in this way is an optimal edit transcript.
6. In Part I we discussed the exact matching problem when don't-care symbols are allowed. Formalize the edit distance problem when don't-care symbols are allowed in both strings, and show how to handle them in the dynamic programming solution.
7. Prove Theorem 11.3.4 showing that the pointers in the dynamic programming table completely capture all the optimal alignments.
8. Show how to use the optimal (global) alignment value to compute the edit distance of two strings and vice versa. Discuss in general the formal relationship between edit distance and string similarity. Under what circumstances are these concepts essentially equivalent, and when are they different?
9. The method discussed in this chapter to construct an optimal alignment left back-pointers while filling in the dynamic programming (DP) table, and then used those pointers to trace back a path from cell (n. m) to cell (0, 0). However, there is an alternate approach that works even if no pointers are available. If given the full DP table without pointers, one can construct an alignment with an algorithm that "works through" the table in a single pass from cell (n, m) to cell (0, 0). Make this precise and show it can be done as fast as the algorithm that fills in the table.
10. For most kinds of alignments (for example, global alignment without arbitrary gap weights), the traceback using pointers (as detailed in Section 11.3.3) runs in $O(n+m)$ time, which is less than the time needed to fill in the table. Determine which kinds of alignments allow this speedup.
11. Since the traceback paths in a dynamic programming table correspond one-to-one with the optimal alignments, the number of distinct cooptimal alignments can be obtained by computing the number of distinct traceback paths. Give an algorithm to compute this number in $O(nm)$ time.

    Hint: Use dynamic programming.
12. As discussed in the previous problem, the cooptimal alignments can be found by enumerating all the traceback paths in the dynamic programming table. Give a backtracking method to find each path, and each cooptimal alignment, in $O(n+m)$ time per path.
13. In a dynamic programming table for edit distance, must the entries along a row be

has already begun or whether a new gap is being started (either opposite character i of S_1 or opposite character j of S_2). This insight, as usual, is formalized in a set of recurrences.

The recurrences

For the case where end gaps are included in the alignment value, the base case is easily seen to be

$$V(i,0) = E(i,0) = -W_g - iW_s,$$
$$V(0,j) = F(0,j) = -W_g - jW_s,$$

so that the zero row and columns of the table for V can be filled in easily. When end gaps are free, then $V(i,0) = V(0, j) = 0$,

The general recurrences are

$$V(i,j) = \max[E(i,j), F(i, j), G(i,j)],$$
$$G(i,j) = \begin{cases} V(i-1, j-1) + W_m, \text{ if } S_1(i) = S_2(j) \\ V(i-1, j-1) - W_{ms}, \text{ if } S_1(i) \neq S_2(j), \end{cases}$$
$$E(i,j) = \max[E(i, j-1), V(i, j-1) - W_g] - W_s,$$
$$F(i,j) = \max[F(i-1, j), V(i-1, j) - W_g] - W_s.$$

To better understand these recurrences, consider the recurrence for $E(i, j)$. By definition, S_1(i) will be aligned to the left of S_2(j). The recurrence says that either 1. $S_1(i)$ is exactly one place to the left of $S_2(j)$, in which case a gap begins in S_1 opposite character $S_2(j)$, and $E(i, j) = V(i, j-1) - W_g - W_s$ or 2. $S_1(i)$ is to the left of $S_2(j-1)$, in which case the same gap in S_1 is opposite both $S_2(j-1)$ and $S_2(j)$, and $E(i, j) = E(i, j-1) - W_s$. An explanation for $F(i, j)$ is similar, and $G(i, j)$ is the simple case of aligning $S_1(i)$ opposite $S_2(j)$.

As before, the value of the optimal alignment is found in cell (n,m) if right end spaces contribute to the objective function. Otherwise the value of the optimal alignment is the maximum value in the nth row or mth column.

The reader should be able to verify that these recurrences are correct but might wonder why $V(i, j-1)$ and not $G(i, j-1)$ is used in the recurrence for $E(i, j)$. That is, why is $E(i, j)$ not $\max[E(i, j-1), G(i, j-1) - W_g] - W_s$? This recurrence would be incorrect because it would not consider alignments that have a gap in S_2 bordered on the left by character $j-1$ of S_2 and ending opposite character i of S_1, followed immediately by a gap in S_1. The expanded recurrence $E(i, j) = \max[E(i, j-1), G(i, j-1) - W_g, V(i, j-1) - W_g] - W_s$ would allow for all alignments and would be correct, but the inclusion of the middle term $(G(i, j-1) - W_g)$ is redundant because the last term $(V(i, j-1) - W_g)$ includes it.

Time analysis

Theorem 11.8.2. The optimal alignment with *affine* gap weights can be *computed* in $O(nm)$ time, the same time asfur optimal alignment without a gap term.

PROOF Examination of the recurrences shows that for any pair (i, j), each of the terms $V(i, j)$, $E(i, j)$, $F(i, j)$, and $G(i, j)$ is evaluated by a constant number of references to previously computed values, arithmetic operations, and comparisons. Hence $O(nm)$ time suffices to fill in all the (n + 1) x (m + 1) cells in the dynamic programming table. □

do not contribute to the cost of the alignment. Show how to use the affine gap recurrences developed in the text to solve the end-gap free version of the affine gap model of alignment. Then consider using the alternate recurrences developed in the previous exercise. Both should run in $O(nm)$ time. Is there any advantage to using one over the other of these recurrences?

29. Show how to extend the ***agrep*** method of Section 4.2.3 to allow character insertions and deletions.

30. Give a ***simple*** algorithm to solve the local alignment problem in $O(nm)$ time if no spaces are allowed in the local alignment.

31. **Repeated substrings.** Local alignment between two different strings finds pairs of substrings from the two strings that have high similarity. It is also important to find substrings of a single string that have high similarity. Those substrings represent inexact ***repeated substrings.*** This suggests that to find inexact repeats in a single string one should locally align of a string against itself. But there is a problem with this approach. If we do local alignment of a string against itself, the best substring will be the entire string. Even using all the values in the table, the best path to a cell (i, j) for $i \neq j$ may be strongly influenced by the main diagonal. There is a simple fix to this problem. Find it. Can your method produce two substrings that overlap? Is that desirable? Later in Exercise 17 of Chapter 13, we will examine the problem of finding the most similar ***nunoverlapping*** substrings in a single string.

32. **Tandem repeats.** Let P be a pattern of length n and T a text of length m. Let P^m be the concatenation of P with itself m times, so P^m has length mn. We want to compute a local alignment between P^m and T. That wilt find an interval in T that has the best global alignment (according to standard alignment criteria) with some tandem repeat of P. This problem differs from the problem considered in Exercise 4 of Chapter 1, because errors (mismatches and insertions and deletions) are now allowed. The particular problem arises in studying the secondary structure of proteins that form what is called a ***coiled-coil***[158]. In that context, P represents a ***motif*** or ***domain*** (a pattern for our purposes) that can repeat in the protein an unknown number of times, and T represents the protein. Local alignment between P^m and T picks out an interval of T that "optimally" consists of tandem repeats of the motif (with errors allowed). If P^m is explicitly created, then standard local alignment will solve the problem in $O(nm^2)$ time. But because P^m consists of identical copies of P, an $O(nm)$-time solution is possible. The method essentially simulates what the dynamic programming algorithm for local alignment would do if it were executed with P^m and T explicitly. Below we outline the method.

The dynamic programming algorithm will fill in an $m + 1$ by $n + 1$ table V, whose rows are numbered 0 to n, and whose columns are numbered 0 to m. Row 0 and column 0 are initialized *to* all 0 entries. Then in each row i, from 1 to m, the algorithm does the following: It executes the standard local alignment recurrences in row i; it sets $V(i, 0)$ to $V(i, n)$; and then it executes the standard local alignment recurrences in row i again. After completely filling in each row, the algorithm selects the cell with largest V value, as in the standard solution to the local alignment problem.

Clearly, this algorithm only takes $O(nm)$ time. Prove that it correctly finds the value of the optimal local alignment between P^m and T. Then give the details of the traceback to construct the optimal local alignment. Discuss why P was (conceptually) expanded to P^m and not a longer or shorter string.

33. **a.** Given two strings S_1 and S_2 (*of* lengths n and m) and a parameter δ, show how to construct the following matrix in $O(nm)$ time: $\mathcal{M}(i, j) = 1$ if and only if there is an alignment *of* S_1 and S_2 in which characters $S_1(i)$ and $S_2(j)$ are aligned with each other and the value of the

nondecreasing? What about down a column or down a diagonal of the table? Now discuss the same questions for optimal global alignment.

14. Give a complete argument that the formula in Theorem 11.6.1 is correct. Then provide the details for how to find the longest common subsequence, not just its length, using the algorithm for weighted edit distance.

15. As shown in the text, the longest common subsequence problem can be solved as an optimal alignment or similarity problem. It can also be solved as an operation-weight edit distance problem.

    Let u represent the length of the longest common subsequence of two strings of lengths n and m. Using the operation weights of $d = 1$, $r = 2$, and $e = 0$, we claim that $D(n, m) = m + n - 2u$ or $u = (m + n - D(n, m))/2$. So, $D(n, m)$ is minimized by maximizing u. Prove this claim and explain in detail how to find a longest common subsequence using a program for operation-weight edit distance.

16. Write recurrences for the longest common subsequence problem that do not use weights. That is, solve the lcs problem more directly, rather than expressing it as a special case of similarity or operation-weighted edit distance.

17. Explain the correctness of the recurrences for similarity given in Section 11.6.1.

18. Explain how to compute edit distance (as opposed to similarity) when end spaces are free.

19. Prove the one-to-one correspondence between shortest paths in the edit graph and minimum weight global alignments.

20. Show in detail that the end-space free variant of the similarity problem is correctly solved using the method suggested in Section 11.6.4.

21. Prove Theorem 11.6.2, and show in detail the correctness of the method presented for finding the shortest approximate occurrence of P in T ending at position j.

22. Explain how to use the dynamic programming table and traceback to find all the optimal solutions (pairs of substrings) to the local alignment problem for two strings S_1 and S_2.

23. In Section 11.7.3, we mentioned that the dynamic programming table is often used to identify pairs of substrings of high similarity, which may not be optimal solutions to the local alignment problem. Given similarity threshold t, that method seeks to find pairs of substrings with similarity value t or greater. Give an example showing that the method might miss some qualifying pairs of substrings.

24. Show how to solve the alphabet-weight alignment problem with affine gap weights in $O(nm)$ time.

25. The discussions for alignment with gap weights focused on how to compute the values in the dynamic programming table and did not detail how to construct an optimal alignment. Show how to augment the algorithm so that it constructs an optimal alignment. Try to limit the amount of additional space required.

26. Explain in detail why the recurrence $E(i, j) = \max[E(i, j-1), G(i, j-1) - W_g, V(i, j-1) - W_g] - W_s$ is correct for the affine gap model, but is redundant, and that the middle term $(G(i, j-1) - W_g)$ can be removed.

27. The recurrences relations we developed for the affine gap model follow the logic of paying $W_g + W_s$ when a gap is "initiated" and then paying W_s for each additional space used in that gap. An alternative logic is to pay $W_g + W_s$ at the point when the gap is "completed." Write recurrences relations for the affine gap model that follow that logic. The recurrences should compute the alignment in $O(nm)$ time. Recurrences of this type are developed in [166].

28. In the end-gap free version of alignment, spaces and gaps at either end of the alignment

Usually a scoring matrix is used to score matches and mismatches, and a affine (or linear) gap penalty model is also used. Experiments [51, 447] have shown that the success of this approach is very sensitive to the exact choice of the scoring matrix and penalties. Moreover, it has been suggested that the gap penalty must be made higher in the substrings forming the a and β regions than in the rest of the string (for example, see [51] and [296]). That is, no fixed choice for gap penalty and space penalty (gap initiation and gap extension penalties in the vernacular of computational biology) will work. Or at least, having a higher gap penalty in the secondary regions will more likely result in a better alignment. High gap penalties tend to keep the α and β regions unbroken. However, since insertions and deletions do definitely occur in the loops, gaps in the alignment of regions outside the core should be allowed.

This leads to the following alignment problem: How do you modify the alignment model and penalty structure to achieve the requirements outlined above? And, how do you find the optimal alignment within those new constraints?

Technically, this problem is not very hard. However, the application to deducing secondary structure is very important. Orders of magnitude more protein sequence data are available than are protein structure data. Much of what is "known" about protein structure is actually obtained by deductions from protein sequence data. Consequently, deducing structure from sequence is a central goal.

A multiple alignment version of this structure prediction problem is discussed in the first part of Section 14.10.2.

37, Given two strings S_1 and S_2 and a text T, you want to find whether there is an occurrence of S_1 and S_2 interwoven (without spaces) in T. For example, the strings abac and bbc occur interwoven in cabbabccdw. Give an efficient algorithm for this problem. (It may have a relationship to the longest common subsequence problem.)

38. As discussed earlier in the exercises of Chapter 1, bacteria! DNA is often organized into circular molecules. This motivates the following problem: Given two linear strings of lengths n and m, there are n circular shifts of the first string and m circular shifts of the second string, and so there are nm pairs of circular shifts. We want to compute the global alignment for each of these nm pairs of strings. Can that be done more efficiently than by solving the alignment problem from scratch for each pair? Consider both worst-case analysis and "typical" running time for "naturally occurring" input.

Examine the same problem for local alignment.

39. **The stuttering subsequence problem [328].** Let P and T be strings of n and m characters each. Give an $O(m)$-time algorithm to determine if P occurs as a subsequence of T.

Now let P^i denote the string P where each character is repeated i times. For example, if $P = abc$ then P^3 is aaabbbccc. Certainly, for any fixed i, one can test in $O(m)$ time whether P^i occurs as a subsequence of T. Give an algorithm that runs in $O(m \log m)$ time to determine the largest i such that P^i is a subsequence of T. Let $Maxi(P, T)$ denote the value of that largest ***i.***

Now we will outline an approach to this problem that reduces the running time from $O(m \log m)$ to $O(m)$. You will fill in the details.

For a string T, let d be the number of distinct characters that occur in T. For string T and character x in T, define $odd(x)$ to be the positions of the odd occurrences of x in T, that is, the positions of the first, third, fifth, etc. occurrence of x in T. Since there are d distinct characters in T, there are d such odd sets. For example, if T = 0120002112022220110001 then $odd(1)$ is 2,9,18. Now define $half(T)$ as the subsequence of T that remains after removing all the characters in positions specified by the d odd sets. For example, $half(T)$

alignment is within δ of the maximum value alignment of S_1 and S_2. That is, if $V(S_1, S_2)$ is the value of the optimal alignment, then the best alignment that puts $S_1(i)$ opposite $S_2(j)$ should have value at least $V(S_1, S_2) - \delta$. This matrix M is used [490] to provide ***some*** information, such as common or uncommon features, about the set of ***suboptimal*** alignments of S_1 and S_2. Since the biological significance of the ***optimal*** alignment is sometimes uncertain, and optimality depends on the choice of (often disputed) weights, it is useful to efficiently produce or study a set of suboptimal (but close) alignments in addition to the optimal one. How can the matrix ***M*** be used to produce or study these alignments?

b. Show how to modify matrix M so that $\mathcal{M}(i, j) = 1$ if and only if $S_1(i)$ and $S_2(j)$ are aligned in every alignment of S_1 and S_2 that has value at least $V(S_1, S_2) - \delta$. How efficiently can this matrix be computed? The motivation for this matrix is essentially the same as for the matrix described in the preceding problem and is used in [443] and [445].

34. Implement the dynamic programming solution for alignment with a gap term in the objective function, and then experiment with the program to find the right weights to solve the cDNA matching problem.

35. The process by which intron-exon boundaries (called ***splice sites)*** are found in mRNA is not well understood. The simplest hope – that splice sites are marked by patterns that always occur there and never occur elsewhere – is false. However, it is true that certain short patterns very frequently occur at the splice sites of introns, In particular, most introns start with the dinucleotide GT and end with ***AG.*** Modify the dynamic programming recurrences used in the cDNA matching problem to enforce this fact.

There are additional pattern features that are known about introns. Search a library to find information about those conserved features – you'll find a lot of interesting things while doing the search.

36. Sequence to structure deduction via alignment

An important application for aligning protein strings is to deduce unknown secondary structure of one protein from known secondary structure of another protein. From that secondary structure, one can then try to determine the three-dimensional structure of the protein by model building methods. Before describing the alignment exercise, we need some background on protein structure.

A string of a typical globular protein (a typical enzyme) consists of substrings that form the tightly wrapped core of the protein, interspersed by substrings that form *loops* on the exterior of the protein. There are essentially three types of secondary structures that appear in globular proteins: a-helixes and β-sheets, which make up the core of the protein, and loops on the exterior of the protein. There are also turns, which are smaller than loops. The structure of the core of the protein is highly conserved over time, so that any large insertions or deletions are much more likely to occur in the loops than in the core.

Now suppose one knows the secondary (or three-dimensional) structure of a protein from one species, and one has the ***sequence*** of the homologous protein from another species, but not its two- or three-dimensional structure. Let S_1 denote the string for the first protein and S_2 the second. Determining two- or three-dimensional structure by crystallography or NMR is very complex and expensive. Instead, one would like to use sequence alignment of S_1 and S_2 to identify the α and β structures in S_2. The hope is that with the proper alignment model, scoring matrix, and gap penalties, the substrings of the a and β structures in the two strings will align with each other. Since the locations of the α and β regions are known in S_1, a "successful" alignment will identify the α and β regions in S_2. Now, insertions and deletions in core regions are rare, so an alignment that successfully identifies the α and β regions in S_2 should not have large gaps in the α and β regions in S_1. Similarly, the alignment should not have large gaps in the substrings of S_2 that align to the known a and β regions of S_1.

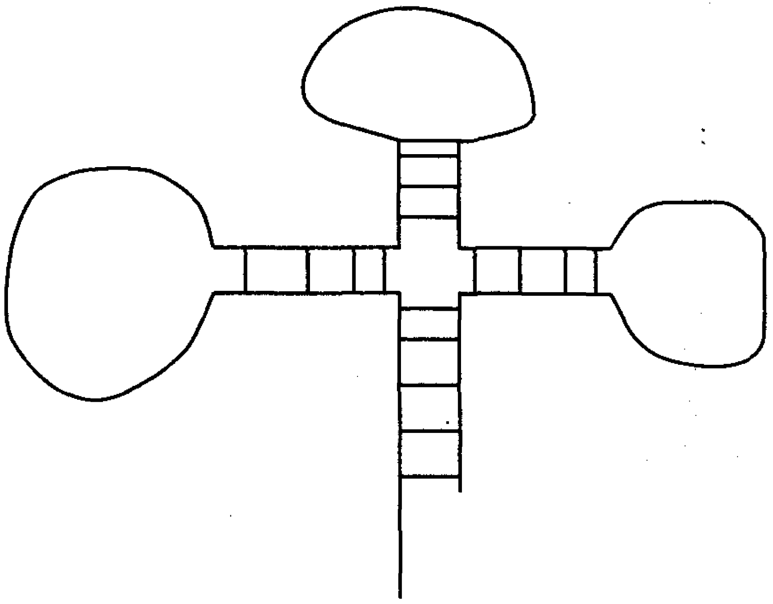

Figure 11.10: A rough drawing of a cloverleaf structure. Each of the small horizontal or vertical lines inside a stem represents a base pairing of $a-u$ or $c-g$.

42. Transfer RNA (tRNA) molecules have a distinctive planar secondary structure called the cloverleaf structure. In a cloverleaf, the string is divided into alternating stems and loops (see Figure 11.10). Each stem consists of two parallel substrings that have the property that any pair of opposing characters in the stem must be complements (a with u; c with g). Chemically, each complementary stem pair forms a bond that contributes to the overall stability of the molecule. A $c-g$ bond is stronger than an a – u bond.

    Relate this (very superficial) description of tRNA secondary structure to the weighted nested pairing problem discussed above.

43. The true bonding pattern of complementary bases (in the stems) of tRNA molecules mostly conforms to the noncrossing condition in the definition of a nested pairing. However, there are exceptions, so that when the secondary structure of known tRNA molecules is represented by lines through the circle, a few lines may cross. These violations of the noncrossing condition are called psuedoknots.

    Consider the problem of finding a maximum cardinality proper pairing where a fixed number of psuedoknots are allowed. Give an efficient algorithm for this problem, where the complexity is a function of the permitted number of crossings.

44. **RNA sequence and structure alignment.** Because of the nested pairing structure of RNA, it is easy to incorporate some structural considerations when aligning RNA strings. Here we examine alignments of this kind.

    Let P be an RNA pattern string with a known pairing structure, and let T be a larger RNA text string with a known pairing structure. To represent pairing structure in P, let $O_P(i)$ be the *offset* (positive or negative) of the mate of the character at position i, if any. For example, if the character at position 17 is mated to the character at position 46, then $O_P(17) = 29$ and $O(46) = -29$. If the character at position i has no mate, then $O_P(i)$ is zero. The structure of T is similarly represented by an offset vector O_T. Then P exactly occurs in T starting at position j if and only if $P(i) = T(j+i-1)$ and $O_P(i) = O_T(j+i-1)$, for each position i in P.

    a. Assuming the lengths of P and T are n and m, respectively, give an $O(n+m)$-time algorithm to find every place that P exactly occurs in T.

    b. Now consider a more liberal criteria for deciding that P occurs in T starting at position j. We again require that $P(i) = T(j+i-1)$ for each position i in P, but now only require that $O_P(i) = O_T(j+i-1)$ when $O_P(i)$ is not zero.

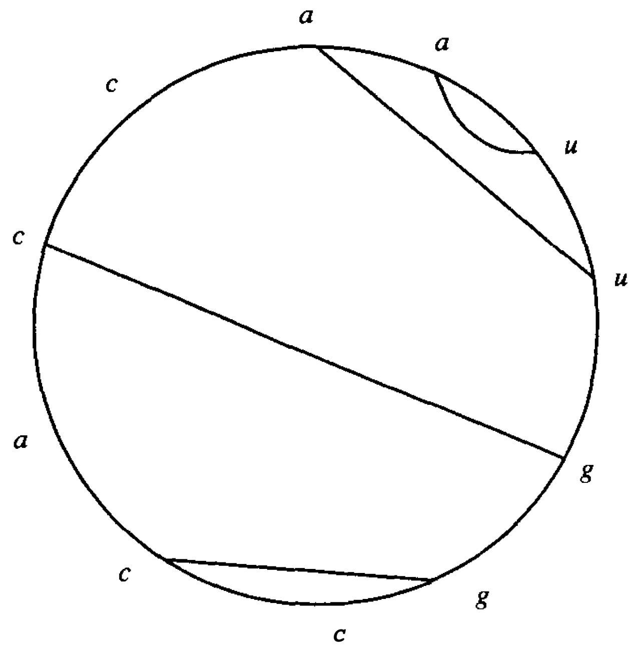

Figure 11.9: A nested pairing, not necessarily of maximum cardinality.

above is 0021220101. Assuming that the number of distinct symbols, d, is fixed ahead of time, give an $O(m)$-time algorithm to find $half(T)$. Now argue that the length of $half(T)$ is at most $m/2$. This will be used later in the time analysis.

Now prove that $|\text{Maxi}(P, T) - 2\text{Maxi}(P, half(T))| \leq 1$.

This fact is the critical one in the method.

The above facts allow us to find Maxi(P, T) in $O(m)$ time by a divide-and-conquer recursion. Give the details of the method: Specify the termination conditions of the divide and conquer, prove correctness of the method, set up a recurrence relation to analyze the running time, and then solve the relation to obtain an $O(m)$ time bound.

Harder problem: What is a realistic application for the stuttering subsequence problem?

40. As seen in the previous problem, it is easy to determine if a single pattern P occurs as a subsequence in a text T. This takes $O(m)$ time. Now consider the problem of determining if any pattern in a set of patterns occurs in a text. If n is the length of all the patterns in the set, then $O(nm)$ **time** is obtained by solving the problem for each pattern separately. Try for a time bound that is significantly better than $O(nm)$. Recall that the analogous substring set problem can be solved in $O(n+m)$ time by Aho–Corasik or suffix tree methods.

41. The **tRNA folding** problem. The following is an extremely crude version of a problem that arises in predicting the secondary (planar) structure of transfer RNA molecules. Let S be a string of n characters over the RNA alphabet a, c, u, g. We define a pairing as set of disjoint pairs of characters in S. A pairing is called proper if it only contains (a,u) pairs or (c, g) pairs. This constraint arises because in RNA a and u are complementary nucleotides, as are c and g. If we draw S as a circular string, we define a *nested pairing* as a proper pairing where each pair in the pairing is connected by a line inside the circle, and where the lines do not cross each other. (See Figure 11.9). The problem is to find a nested pairing of largest cardinality. Often one has the additional constraint that a character may not be in a pair with either of its two immediate neighbors. Show how to solve this version of the tRNA folding problem in $O(n^3)$ time using dynamic programming.

Now modify the problem by adding weights to the objective function so that the weight of an $a-u$ pair is different than the weight of a $c-g$ pair. The goal now is to find a nested pairing of maximum total weight. Give an efficient algorithm for this weighted problem.

two adjacent gaps where each is in a different string. For example, the alignment

$$\begin{array}{ccccccc} x & x & a & b & c & y & y \\ x & x & i & d & e & y & y \end{array}$$

would never be found by these modified recurrences.

There seems no modeling justification to prohibit adjacent gaps in opposite strings. In fact some mutations, such as ***substring inversions*** (which are common in DNA), would be best represented in an alignment **as** adjacent gaps of this type, unless the model of alignment has an explicit notion of inversion (we will consider such a model in Chapter 19). Another example where adjacent spaces would be natural occurs when comparing two mRNA strings that arise from alternative intron splicing. In eukaryotes, genes are often comprised of alternating regions of exons and introns. In the normal mode of transcription, every intron is eventually spliced out, so that the mRNA molecule reflects a concatenation of the exons. But it can also happen, in what is called ***alternative splicing,*** that exons can be spliced out as well as introns. Consider then the situation where all the introns plus exon i are spliced out, and the situation where all the introns plus exon $i+1$ are spliced out. When these two mRNA strings are compared, the best alignment may very well put exon i against a gap in the second string, and then put exon $i+1$ against a gap in the first string. In other words, the informative alignment ***would*** have two adjacent gaps in alternate strings. In that case, the recurrences above do not correctly implement the second viewpoint.

Write recurrences for arbitrary gap weights to allow adjacent gaps in the two opposite strings and yet prohibit adjacent gaps in a single string.

Give an efficient algorithm to find all locations where P occurs in T under the more liberal definition of occurrence. The naive, $O(nm)$-time solution of explicitly aligning P to every starting position j and then checking for a match is not efficient. An efficient solution can be obtained using only methods in Part I and II of the book.

c. Discuss when the more liberal definition is reasonable and when it may not be.

45. A gap modeling question

The recurrences given in Section 11.8.5 for the case of arbitrary gap weights raise a subtle question about the proper gap model when the gap penalty w is arbitrary. With those recurrences, any single gap can be considered as two or more gaps that just happen to be adjacent. Suppose, for example, $w(q) = 1$ for $q \le 5$, and $w(q) = 10^6$ for $i > 5$. Then, a gap of length 10 would have weight 10^6 if considered as a single gap, but would only have weight 2 if considered as two adjacent gaps of length five each. The recurrences from Section 11.8.5 would treat those ten spaces as two adjacent gaps with total weight 2. Is this the proper gap model?

There are two viewpoints on this question. In one view, the goal is to model the most likely set of mutation events transforming one string into another, and the alignment is just an aid in displaying this transformation. The primitive mutational events allowed are the transformation of single characters (mismatches in the alignment) and insertion and deletion of blocks of characters of arbitrary lengths (each of which causes a gap in the alignment). With this view, it is perfectly proper to have two adjacent gaps on the same string. These are just two block insertions or deletions that happen to have occurred next to each other. If the gap weights correctly model the costs of such block operations, and the cost is a concave increasing function of length as in the above example, then it is much more likely that a long gap will be created by several insertion or deletion events than by a single such event. With this view, one should insist that the dynamic program allow adjacent gaps when they are advantageous.

In the other view, one is just interested in how "similar" two strings are, and there may be no explicit mutational model. Then, a given alignment of two strings is simply one way to demonstrate the similarity of the two strings. In that view, a gap is a maximal set of adjacent spaces and so should not be broken into smaller gaps.

With arbitrary gap weights, the dynamic programming recurrences presented correctly model the first view, but not the second. Also, in the case of convex (and hence affine or constant) gap weights, the given recurrences correctly model both views, since there is no incentive to break up a gap into shorter gaps. However, if gap weights with concave increasing sections are thought proper, then different recurrences are required to correctly model the second view, The recurrences below correctly implement the second view:

$$V(i, j) = \max[E(i, j), F(i, j), G(i, j)],$$

$$G(i, j) = V(i-1, j-1) + s(S_1(i), S_2(j)),$$

$$E(i, j) = \max[G(i, k) - w(j-k)] \text{ (over } 0 \le k \le j-1),$$

$$F(i, j) = \max[G(l, j) - w(i-l)] \text{ (over } 0 \le l \le i-1).$$

These equations differ from the recurrences of Section 11.8.5 by the change of $V(i, k)$ and $V(l, j)$ to $G(i, k)$ and $G(l, j)$ in the equations for $E(i, j)$ and $F(i, j)$, respectively. The effect is that every gap except the left-most one must be preceded by two aligned characters; hence there cannot be two adjacent gaps in the same string. However, this also prohibits

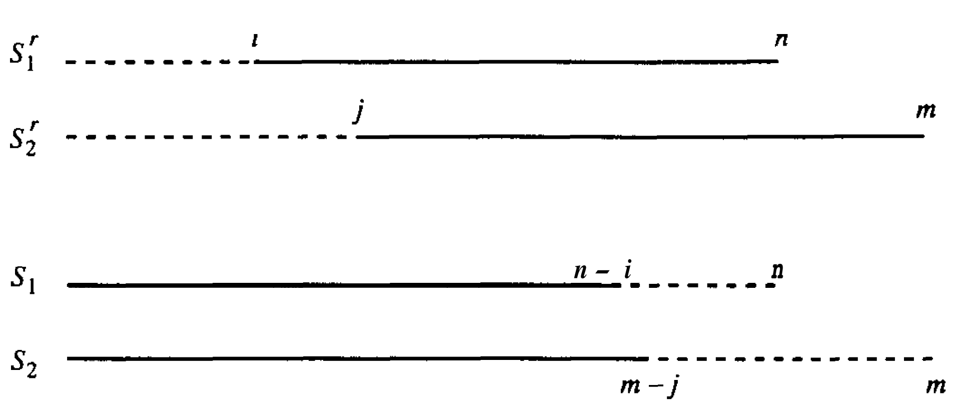

Figure 12.1: The similarity of the first i characters of S_1^r and the first j characters of S_2^r equals the similarity of the last i characters of S_1 and the last j characters of S_2. (The dotted lines denote the substrings being aligned.)

single row of the full table can be found and stored in those same time and space bounds. This ability will be critical in the method to come.

As a further refinement of this idea, the space needed can be reduced to one row plus one additional cell (in addition to the space for the strings). Thus $m + 1$ space is all that is needed. And, if $n < m$ then space use can be further reduced to $n + 1$. We leave the details as an exercise.

12.1.2. How to find the optimal alignment in linear space

The above idea is fine if we only want the similarity $V(n, m)$ or just want to store one preselected row of the dynamic programming table. But what can we do if we actually want an alignment that achieves value $V(n, m)$? In most cases it is such an alignment that is sought, not just its value. In the basic algorithm, the alignment would be found by traversing the pointers set while computing the full dynamic programming table for similarity. However, the above linear space method does not store the whole table and linear space is insufficient to store the pointers.

Hirschberg's high-level scheme for finding the optimal alignment in only linear space performs several smaller alignment computations, each using only linear space and each determining a bit more about an actual optimal alignment. The net result of these computations is a full description of an optimal alignment. We first describe how the initial piece of the full alignment is found using only linear space.

Definition For any string α, let α^r denote the reverse of string α.

Definition Given strings S_1 and S_2, define $V^r(i, j)$ as the similarity of the string consisting of the first i characters of S_1^r, and the string consisting of the first j characters of S_2^r. Equivalently, $V^r(i, j)$ is the similarity of the last i characters of S_1 and the last j characters of S_2 (see Figure 12.1).

Clearly, the table of $V^r(i, j)$ values can be computed in $O(nm)$ time, and any single preselected row of that table can be computed and stored in $O(nm)$ time using only $O(m)$ space.

The initial piece of the full alignment is computed in linear space by computing $V(n, m)$ in two parts. The first part uses the original strings; the second part uses the reverse strings. The details of this two-part computation are suggested in the following lemma.

Lemma 12.1.1. $V(n, m) = \max_{0 \le k \le m}[V(n/2, k) + V^r(n/2, m - k)]$.